Generative Geometric Design and Boundary Solid Grammars

Dissertation

by

Jeff A. Heisserman

Department of Architecture
Carnegie Mellon University
Pittsburgh, Pennsylvania

May 1991

Carnegie Mellon University

Department of Architecture
College of Fine Arts

Dissertation

SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

TITLE *Generative Geometric Design*

and Boundary Solid Grammars

PRESENTED BY *Jeff A. Heisserman*

ACCEPTED BY ADVISORY COMMITTEE

Robert F. Woodbury	PRINCIPAL ADVISOR	DATE
Dana S. Scott	ADVISOR	DATE
Ramesh Krishnamurti	ADVISOR	DATE
Ulrich Flemming	ADVISOR	DATE

Carnegie Mellon University

Department of Architecture
College of Fine Arts

Dissertation

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

TITLE *Generative Geometric Design*
and Boundary Solid Grammars

PRESENTED BY *Jeff A. Heisserman*

APPROVED BY

David Lowry Burgess DEAN DATE

John P. Eberhard DEPARTMENT HEAD DATE

Robert F. Woodbury ADVISOR DATE

Abstract

This thesis explores the automatic generation of solid models based on a grammatical paradigm. It introduces a formalism, *boundary solid grammars*, for this purpose. In this formalism, a set of geometric rules is applied to an initial solid model to generate a language or family of solids. A rule may match on a portion of the boundary of a solid, and then modify the solid or add new solids.

Genesis is presented as an implementation of the formalism. A number of grammars have been constructed to demonstrate the concepts and usefulness of the formalism. These grammars generate simple geometric forms including snowflakes, recursive octahedra, “fractal” mountains, and spirals. Another grammar generates stereo lithography support structures. Queen Anne houses have been characterized with a more extensive grammar. Grammars are also being developed to generate housings for small computers and structural designs for high rise buildings.

The thesis introduces the *unary shape operations* and a new paradigm for solid modeling. The unary shape operations take models that may have self-intersections, interpret the models considering the given geometry and face orientations, and produce valid models. Local operations, the unary shape operations, and Boolean operations are used together within a valid modeling scheme.

The thesis introduces a new boundary representation for manifold and nonmanifold solids, the *generalized split-edge representation*. It describes generalized Euler operations which manipulate the topology of the nonmanifold representation. Finally, the thesis presents a form of the Euler-Poincaré equation that characterizes the relationship between elements of nonmanifold surfaces of solids.

For Mary and Amie

0.1 Preface

This thesis is the result of a rather personal exploration into the area of grammar-based generation of geometric models. It includes a number of different but related topics. Although it may be obvious how these research topics are related to each other, it may not be so obvious why I chose to explore this particular set of topics. The reasoning behind this choice of topics is presented below.

My original intent was to focus on rules and grammars that operate on a solid modeling representation. It became clear from my analysis of the shape grammar literature and experience with expert systems that dealt with (ad hoc) geometry that it is important to work with geometrically complete representations (i.e. solid models), and that rules are able to match and operate on boundary elements. This suggested using a 2-manifold boundary representation of solids. Rules match on topological elements with conditions on the geometry, and modify the solid representations with local operations on their geometry and topology.

The local operations allow the possibility of invalid results due to self intersections of the bounding surfaces. To address this problem, I began playing with the unary union operation. The unary union takes a model that may have self-intersections, interprets the model considering the given geometry and face orientations, and produces a valid model. I eventually expanded the concept to the unary intersection operation, and finally to the generalized unary intersection.

The unary shape operations solve the invalidity problem, but expose additional problems. First, the unary operations produce nonmanifold results. Just as the regularized Boolean operations are closed on r-sets, the unary shape operations are closed on boundary representations of r-sets. Secondly, the relationship between the unary shape operations and the regularized Boolean operations was not clear. I found that the Boolean operations form special cases of the unary shape operations, and that the unary shape operations may be used to compute the Boolean operations. My use of local operations and unary shape operations allowed me to define a new modeling scheme.

In order for the representation to be closed under the unary operations, I decided to develop a nonmanifold boundary representation for solids. This representation needed to be both efficient for rule matching, and compatible with the 2-manifold matching and operations already completed. This resulted in the development of the generalized split-edge representation. It also began the exploration of the topological properties of nonmanifold solids, and the development of an extension to the Euler-Poincaré equation. It also required generalized Euler operations to manipulate the nonmanifold adjacencies of the representation.

0.2 Acknowledgements

There are many people who have contributed to this thesis, and made my years at Carnegie Mellon enjoyable and rewarding. Although it is not possible to name all of these people, I would like to name a few that have contributed directly.

My sincere appreciation goes to Robert Woodbury, my advisor and thesis committee chairman, for his perspective and unqualified support throughout the work on this thesis. I would like to thank my thesis committee, Dana Scott, Ramesh Krishnamurti, and Ulrich Flemming, for their stimulating discussions, insight, and advice.

A number of people have contributed to this thesis by providing comments on its various drafts and engaging in useful discussions. I would like to thank all those people, with special thanks to Chris Carlson, Spiro Michaylov, Nevin Heintze, Charles Coffman, Kiumarse Zamanian, Rich Quadrel, and Roy Taylor.

I would like to thank those friends who have contributed to the development of **Genesis** . Clauss Strauch applied his programming expertise to the first graphics routines used in **Genesis** . Mikako Harada developed integral properties routines for her master's project that were later integrated into **Genesis** . Thanks to Spiro Michaylov for his help and expertise with the Monash **CLP**($\mathcal{R}$)interpreter and the IBM **CLP**($\mathcal{R}$)complier. I am especially grateful to Shih Hai Chiang and Ching Yin for their work expanding the set of rules for the Queen Anne grammar, and for their patience and persistence working with earlier, slow, undocumented versions of **Genesis** . Manuel Anaya-Dufresne and Steven Meyer have aided in the process of debugging and developing of **Genesis** , while constructing their grammars for the generation of computer housings and structural designs for high rise buildings. I would also to acknowledge my appreciation to the students in my Geometric Modeling class for their helpful comments and patience working with **Genesis** .

Thanks to the faculty, staff, and students of the Engineering Design Research Center for their contribution to a stimulating research environment. Special thanks to Clauss Strauch for maintaining my computing environment both for system development and text processing. I would like to acknowledge the support of the National Science Foundation for their support under Grant MSM-8717307, and the NSF Engineering Research Program for their support of the Engineering Design Research Center. Thanks goes to the Hewlett-Packard Corporation for their the generous equipment grant, on whose workstations this research was conducted. Thanks also goes to the Intel Corporation for their equipment donation for related work.

Most of all, I would like to thank my wife, Mary, and daughter, Amie, for their love and patience during the long process of research and writing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> The increasing ubiquity of computer graphics
> and three-dimensional images of astonishing
> realism have artfully concealed the fact that
> important aspects of three-dimensional computer
> graphics are hard, and in spots the frontier
> of what is easy has advanced little in 20 years.
> *Robert Sproull (1990)*

Geometric design is central to many areas of engineering, computer graphics, and architecture. Examples of geometric design problems encountered by practitioners in these areas are listed below:

- Compose the rooms of a house according to a particular style and construct a massing model. From this model, construct the roof, floors, ceilings, and interior and exterior walls, and locate doors and windows.
- Given a volumetric model of the bays of a high-rise office building, place the structural columns, beams, shear walls and floor slabs.
- Given a set of components for a computer, lay out the components and design a housing to enclose them.
- Given a model of a part, generate support structures for manufacturing the part using a stereo lithography process.
- Construct a model of a set of dies to manufacture a given part.

Current solid modeling systems provide users with little assistance in accomplishing these tasks. They generally provide operations to create, manipulate, and modify models. To construct the models previously mentioned, a user must examine the model and consider what parts should be

Figure 1.1: A Queen Anne house.

added. The user must then determine where and how to apply the solid modeling operations to create the new components. Programming facilities are often added, however they provide little more than the ability to create designs with parametric variations.

Robert Sproull expressed this in a recent address [49], continuing the quotation above:

> What remains hard is modeling. The structure inherent in three-dimensional models is difficult for people to grasp and difficult too for user interfaces to reveal and manipulate. Only the determined model three-dimensional objects, and they rarely invent a shape at a computer, but only record a shape so that analysis or manufacturing can proceed. The grand challenges in three-dimensional graphics are to make simple modeling easy and to make complex modeling accessible to far more people.

This thesis attempts to address these modeling problems in four ways:

1. solid modeling with local operations and unary shape operations;

2. logical reasoning about solids;

3. rules that match and operate on solids; and

4. grammars generating languages of solids.

First, the thesis introduces a new approach to solid modeling. The approach uses a boundary representation of solids with local operations to modify the topological and geometric representation, and the *unary shape operations* to guarantee valid representations. Local operations allow more intuitive and direct manipulation of solid models. However, local operations may cause a model to have self-intersections; thus, they may be a source of geometric invalidity. The unary shape operations ensure the validity of the boundary representations. These are global operations that take any topologically valid model and interpret the model, considering the given geometry and face orientations, to produce topologically and geometrically valid models.

Second, the thesis reports a method for reasoning about conditions or features of solids using first order logic. The graph based boundary representation of solids determines the basic axioms for reasoning about a set of solids. Clauses are then constructed to express complex conditions. A goal clause may be used to express a condition, and a theorem proving mechanism is then used to determine the satisfiability of the condition.

Third, the thesis introduces rules that match on conditions of the solid models and perform modifications on them. These *solid rules* provide the ability to build "smart" operations that locate where operations should be applied and how to apply them.

Finally, the thesis introduces *boundary solid grammars*, a rule-based formalism for generating complex models of rigid solid objects. Solids are represented by their boundary elements, i.e. vertices, edges, and faces, with coordinate geometry associated with the vertices. Non-geometric data may be associated with any of these elements. Rules are used to match conditions of a solid or collection of solids and then modify them or create additional solids. A boundary solid grammar consists of an initial solid and a set of rules. It produces a language or family of solid models.

The boundary solid grammar formalism and modeling approach are implemented in the **Genesis** boundary solid grammar interpreter. Example grammars generate simple geometric forms including a three-dimensional variant of Koch snowflakes, recursive octahedra, "fractal" mountains, and spirals. Another grammar generates stereo lithography support structures. Queen Anne houses [18] are characterized in a more extensive grammar. Grammars are being developed that generate personal computer housings and structural designs for high rise buildings. A description of the **Genesis** boundary solid grammar interpreter and the example grammars are included in the thesis.

Organization of the Thesis

The dissertation is organized into three parts. The first part, **Geometric Modeling**, presents the solid modeling approach, including mathematical models, representation, and modeling operations used. This part introduces several new developments: a new approach to solids modeling; an exploration of the properties of nonmanifold solids; a new data structure for explicit representation of manifold and nonmanifold solids; Euler operations for nonmanifold solids; and the unary shape operations on boundary representations.

The second part, **Generative Geometric Design**, presents the boundary solid grammar formalism, with explanations of the representation, methods of reasoning about conditions, constructing operators, formulation of solid rules, and the generative power of the grammars. The formalism is illustrated with grammars that generate solid models in several domains. The **Genesis** boundary solid grammar interpreter is also presented.

The final part of the thesis, **Conclusion**, consists of two chapters. The first chapter summarizes the results of this thesis. The second chapter suggests some future directions for research.

Suggestions to the Reader

Different readers will approach this document with different backgrounds, and will be interested in different aspects of it. For the reader who does not wish to read the dissertation from cover to cover, I can offer the following suggestions. Most readers interested in solid modeling should read Chapter 2 for a characterization of mathematical models of solids and their properties, Chapter 3 for a discussion of the boundary representation used, Chapter 4 for local operations on geometry and topology, Chapter 5 for the unary shape operations and Boolean operations, and finally Chapter 6 for a description of the modeling approach. A few suggestions for further work are discussed in the final chapter.

Those interested primarily in generative geometric design should skim Chapters 2 through 6, then read Chapter 7 for the definition of boundary solid grammars, including a discussion of the techniques for reasoning about solids and constructing solid rules, and Chapter 8 for a presentation of grammars that have been developed.

Finally, those readers interested in implementing solid grammar systems will need an understanding of all the geometric modeling and generative design chapters, and should focus on the material in Chapter 9. Those interested in none of these topics should still enjoy the pictures in Chapter 8.

Part I

Geometric Modeling

Solid modeling seeks to provide geometrically complete representations of solid objects. There are two dominant representation schemes in solid modeling: constructive solid geometry, and boundary representation.

In constructive solid geometry a solid is represented by (valid) primitive solids combined using regularized Boolean operations and rigid transformations. The interior and surface of a solid are defined implicitly. By restricting the creation of solids in this fashion, a CSG model is always valid. That is, it always represents subsets of E^3 that are bounded, closed, regular, and semi-analytic.

Boundary representations describe the surfaces of solids by their surface elements, vertices, edges, and faces, and the adjacency relations between these elements. Euler operators are used to maintain the integrity of the adjacency relations when modifying a solid. The geometry of solids may be assigned by the user. Using Euler operations and unrestricted assignment of geometry to modify boundary descriptions may cause the boundary to have self-intersections. In order to ensure that a model is valid, complex and computationally expensive tests are required. However, most systems simply require the user to determine and maintain the validity of models, in deference to simpler user interaction and faster response.

This thesis introduces a new approach to the boundary representation scheme. The Euler operations maintain topological validity (surface neighborhood and orientation conditions), while the assignment of geometry is not restricted. The boundary descriptions may then have self-intersections. Unary shape operations, introduced in the thesis, then modify the topology and geometry to produce valid solids. The unary shape operations find self-intersections in the surfaces of the solid, and remove portions of the boundary that are within the solid.

A new boundary representation is also used, the *generalized split-edge representation.* It allows the representation of topologies of both manifold and nonmanifold solids, and explicitly represents topological adjacencies necessary for quick matching of the bounding elements. This last characteristic is of primary importance to the formalism introduced in the second part of the thesis. Nonmanifold Euler operators are introduced to manipulate the nonmanifold adjacencies of the representation.

The next five chapters discuss the mathematical models and properties of solids, computer representations, modeling operations, and solid modeling representation schemes.

Chapter 2

Modeling Solids

This chapter presents rigorous descriptions of solids and their properties. We begin by presenting a definition for solids, then consider mathematical characterizations of solids, and conditions of topological and geometric validity. Finally, we explore the topological properties of solids.

A solid is an idealized physical object. A solid is rigid, i.e. its shape does not deform when we rotate or translate it. The shape of a solid is independent of its location. A solid must be homogeneously three-dimensional. It cannot have elements of lower dimensionality, such as "dangling" faces, edges, or vertices. It must also have a clearly defined interior, bounding surface, and exterior. A solid must have finite extent. That is, it must occupy a finite portion of space.

2.1 Point Set Models

This section uses concepts from point set topology to characterize mathematical models of solid objects. It assumes some familiarity with the terminology of set theory, and introduces definitions (adapted from Requicha [42], Simmons [48], and Henle [21]) that will be used throughout the thesis. We characterize solid objects as *r-sets*, as introduced by Requicha [41].

Definition 2.1 *Let W be a non-empty set. A class T of subsets of W is called a* **topology** *on W if it satisfies the following two conditions:*

1. *the union of every class of sets in T is a set in T;*
2. *the intersection of every finite class of sets in T is a set in T.*

Definition 2.2 *A* **topological space** *is a pair (W, T) where W is a set and T is a topology on W. The sets in the class T are called* **open sets** *of the topological space, and the elements of W are called its* **points**.

Definition 2.3 *A* **neighborhood** $N(p)$ *of a point* p *is a subset of* X *which contains an open set which contains* p. *If* $N(p)$ *is an open set it is called an* **open neighborhood**.

Definition 2.4 *A point* p *is a* **limit point** *of a subset* X *of* E^3 *if each neighborhood of* p *contains at least one point of* X *different from* p.

Definition 2.5 *A set is* **closed** *if and only if it contains all its limit points.*

Definition 2.6 *The* **closure** *of a subset* X *of* E^3, *denoted* kX, *is the union of* X *with the set of all its limit points.*

Definition 2.7 *A point* p *of* S *is an* **interior point** *of a subset* X *of* S *if* X *is a neighborhood of* p, *i.e., if* X *contains an open set which contains* p.

Definition 2.8 *The* **interior** *of a subset* X *of* W, *denoted* iX, *is the set of all the interior points of* X.

Definition 2.9 *The* **boundary** *of a subset* X *of* W, *denoted* ∂X, *is the set* $kx - iX$.

We consider here closed regular sets.

Definition 2.10 *The* **regularization** *of a subset* X *of* W, *denoted* rX, *is the closure of the interior of* X, $rX = kiX$.

Definition 2.11 *A set* X *is* **regular** *if* $X = rX$, *i.e., if* $X = kiX$

Definition 2.12 *A subset* X *of* E^3 *is* **bounded** *if it can be enclosed in a sphere of finite radius.*

Definition 2.13 *A subset* X *of* E^3 *is* **semi-analytic** *if it can be expressed as a finite combination, via the usual set operations (intersection, union, difference, and complement) of sets of the form*

$$\{(x,y,z)|F_i(x,y,z) \leq 0\}$$

where the F_i *are analytic functions (i.e., they are locally expressible as convergent power series).*

Finally, we can define r-sets.

Definition 2.14 *An* **r-set** *is a subset of* E^3 *that is bounded, closed, regular, and semi-analytic.*

R-sets accurately characterize the points of solid objects. We may also represent solid objects by their bounding surfaces. In the next section, we consider the characteristics of surfaces, with special attention to the surfaces of r-sets.

2.2 Surface Models

The surface of a solid object may be viewed as a smooth, continuous surface. This sort of surface corresponds to the notion of a 2-manifold.

Definition 2.15 *A* **2-manifold** *is a topological space where each point has an open neighborhood homeomorphic to an open disk in* E^2

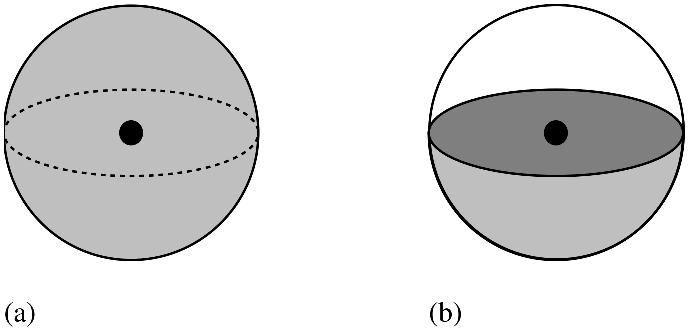


Figure 2.1: The neighborhood of an interior point, and a manifold boundary point.

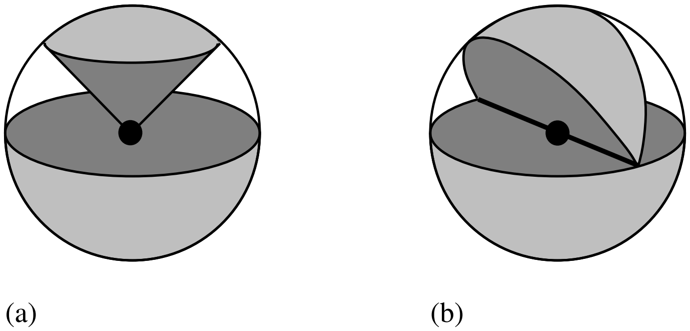


Figure 2.2: Nonmanifold neighborhoods of a vertex, and an edge.

We use the term *manifold solid* to denote a solid with a 2-manifold surface. The neighborhood of a point inside a solid, and a point on a manifold boundary are illustrated in Figure 2.1. Nonmanifold neighborhoods of a vertex, and an edge are presented in Figure 2.2.

For a manifold solid, every point on its surface is two-dimensional. A nonmanifold solid will have points that are not two-dimensional, i.e., that have two or more portions of the surface that touch at an edge or a vertex. Figure 2.3 shows a manifold solid (a) and a nonmanifold solid (b).

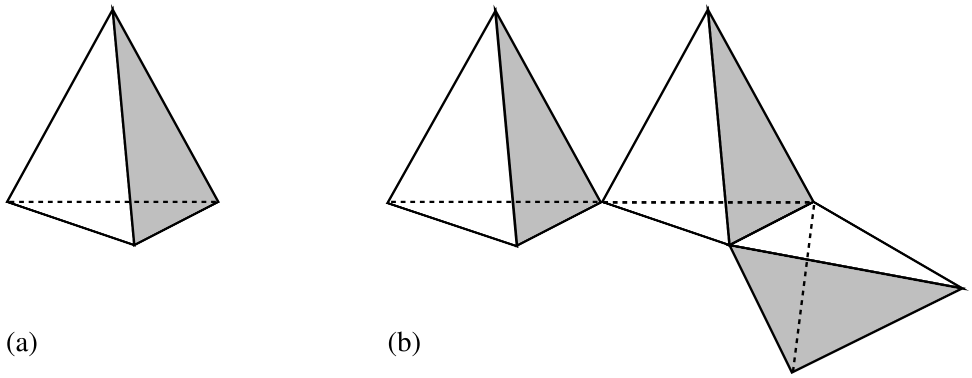


Figure 2.3: A manifold and a nonmanifold solid.

We consider solids whose boundaries are either manifold or nonmanifold surfaces. However, we do not allow lower dimensional (non-solid) elements, such as dangling faces and edges, or isolated vertices.

Using these classifications, we can characterize the adjacencies of manifold solids by the following conditions:

- each vertex has a single cone of edges and faces about it;
- each edge is adjacent to exactly two faces; and
- each edge is adjacent to exactly two vertices.

The adjacencies of nonmanifold solids have a slightly looser set of conditions:

- each vertex has one or more cone of edges and faces about it;
- each edge is adjacent to one or more pairs of faces; and
- each edge is adjacent to exactly two vertices.

It is useful to note that every manifold solid is considered a nonmanifold solid, but not *vice versa.*

We can consider a nonmanifold solid as the *immersion* of several manifold solids [23]. The manifold boundaries of the solids are allowed to intersect, but the intersections are restricted to sets of zero or one dimension.

Topological and Geometric Validity

A solid can be represented unambiguously by describing its surface, and topologically orienting it such that we can tell, at each surface point, on which side the solid interior lies. This description has two parts, a *topological* description of the connectivity and orientation of vertices, edges, and faces, and a *geometric* description for embedding these surface elements in space. The topological description of a solid is generally referred as its *topology* and the geometric description, as its *geometry.* The geometry specifies, for example, the coordinates of the vertices. The topology and geometry together define a boundary representation solid model.

It is possible to construct topologies that do not correspond to solid objects. The topological description may be invalid due to meaningless adjacencies. For example, one can imagine constructing a topology with an adjacency relation between an edge and more than two vertices. This is clearly invalid, since an edge must be bounded by exactly two vertices for any physical solid. Alternately, the topological description could have valid adjacencies, but may not be orientable. For example, it is impossible to orient a Klein bottle. *Euler operators* are used in order to construct only topologically valid models.

A solid is *geometrically valid* if the faces of the solid intersect each other only at common edges of the faces and do not intersect at their internal points, and edges intersect only at the vertices of the edges. A solid representation must be both topologically and geometrically valid in order to be a valid representation of a rigid solid object.

2.3 Topological Properties of Solids

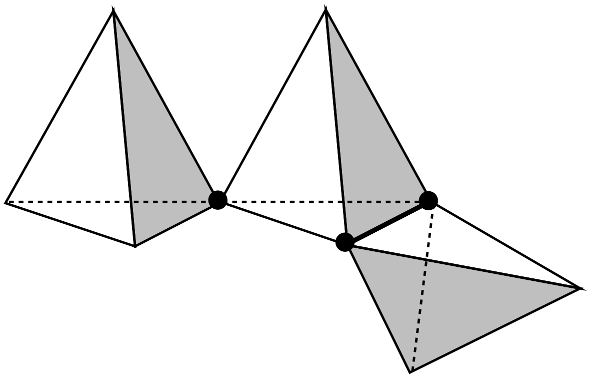


Figure 2.4: A nonmanifold solid.

The Euler-Poincaré equation,

$$v - e + f = 2(s - g), \tag{2.1}$$

relates the numbers of topological elements of 2-manifold surfaces. Here v, e, f, s, and g refer to the numbers of vertices, edges, faces, shells (surfaces) and handles. However, the equation does not correctly relate the elements of nonmanifold surfaces, and specifically the boundaries of r-sets.

We can see this clearly by a simple example. The nonmanifold solid in Figure 2.4 may be formed by joining two tetrahedra at a single vertex, and a third tetrahedron at an edge. The heavy solid vertices and thick edges in the figure indicate vertices and edges with nonmanifold neighborhoods. The solid has 9 vertices, 17 edges, 12 faces, one (nonmanifold) shell, and zero handles. Clearly,

$$9 - 17 + 12 \neq 2(1 - 0).$$

What then is the relation between the elements of regular solids?

We introduce a generalized form of the Euler-Poincaré equation

$$v' - e' + f = 2(s' - g) \tag{2.2}$$

which relates the number of vertex uses, edge uses, faces, shell uses, and (manifold) handles of both 2-manifold and nonmanifold surfaces of r-sets. In addition, we introduce the equation

$$(v' - v) - (e' - e) - (s' - s) = g' - c \tag{2.3}$$

which relates the number of vertex uses, vertices, edge uses, edges, shell uses, and shells to the number of nonmanifold handles and nonmanifold chambers.

To understand these equations, we need definitions of the new terms. We introduce definitions of vertex uses, edge uses, shell uses, nonmanifold handles, and nonmanifold chambers.

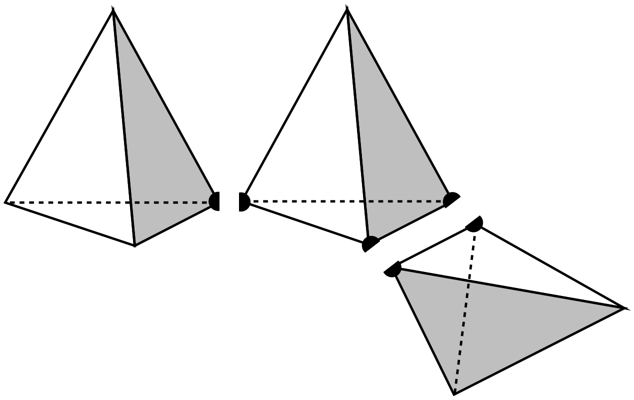


Figure 2.5: Vertex uses.

A *vertex use*, denoted v', is a 2-manifold use of a vertex. This is a use of a vertex with respect to a shell. For example, the three nonmanifold vertices of the solid in Figure 2.4 each have two vertex uses. These as shown in Figure 2.5. Any vertex on a 2-manifold surface will have a single vertex use. A vertex with a nonmanifold neighborhood will have multiple vertex uses.

An *edge use*, denoted e', is a 2-manifold use of an edge, and corresponds to a pair of incident faces with respect to that edge. The two edge uses of the nonmanifold (thickened) edge of the solid

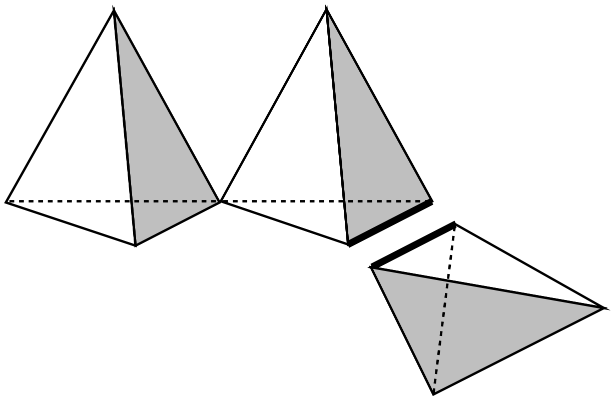


Figure 2.6: Edge uses.

in Figure 2.4 are shown in Figure 2.6. Any edge on a 2-manifold surface will have a single pair of incident faces, thus a single edge use. An edge with a nonmanifold neighborhood will have multiple edge uses.

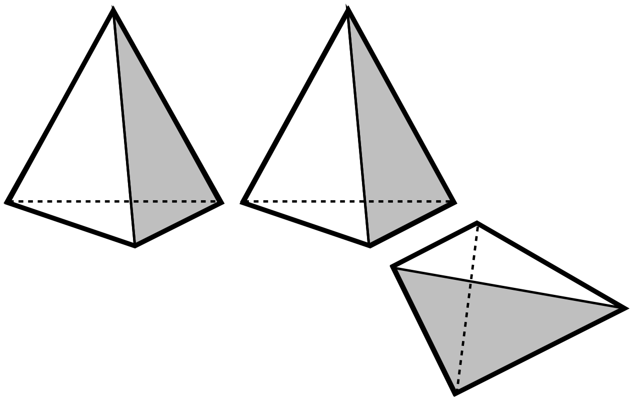


Figure 2.7: Shell uses.

A *shell use*, denoted s', is a 2-manifold piece of a connected surface. We can separate the shell uses of a surface by splitting apart the vertices and edges with nonmanifold neighborhoods. The three shell uses of the nonmanifold solid in Figure 2.4 are illustrated in Figure 2.7. We can define a shell use as a partition of the faces, edge uses, and vertex uses of a shell (surface) such that one can traverse from any face, edge use, or vertex use of that partition to any other by their adjacencies without crossing any faces, edge uses, or vertex uses of any other partition. A 2-manifold shell will have a single shell use, and a 2-manifold solid will have an equal number of shells and shell uses. A nonmanifold solid may have more shell uses than shells.

We can define the first Betti number (connectivity number) of a 2-manifold surface (our definition is adapted from [21]), in order to determine its number of handles and genus. Using a variation of this definition, we can define the connectivity number for a nonmanifold surface. Next, we introduce the concept of a *nonmanifold handle*, and use the new connectivity number to determine the number of nonmanifold handles and the genus of a nonmanifold solid.

Definition 2.16 *The first Betti number h_1 is the maximum size of a set of closed curves that taken together can be drawn on a surface that will not divide the surface into two or more pieces. This is also known as the connectivity number of the surface.*

For any nonmanifold solid, we can create a 2-manifold surface by offsetting the surface of the solid by an arbitrarily small amount ϵ. By choosing a sufficiently small ϵ, we can guarantee that we do not create additional handles or nonmanifold conditions from self-intersection of the surface. By offsetting the surface, we are able to "thicken" the vertices and edges that have nonmanifold neighborhoods, thereby eliminating these nonmanifold conditions. This is used to define the connectivity number for nonmanifold solids.

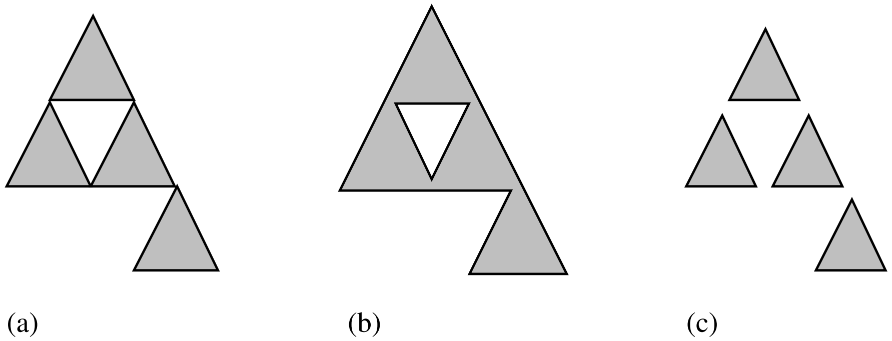


Figure 2.8: A solid, its offset and minus offset solids.

Definition 2.17 *The manifold connectivity number h_1 of the surface of a nonmanifold solid is the maximum size of a set of closed curves that taken together can be drawn on a 2-manifold minus offset surface of that solid (offset from the complement of the solid) that will not divide the surface into two or more pieces.*

Definition 2.18 *The nonmanifold connectivity number h_1' of the surface of a nonmanifold solid is the maximum size of a set of closed curves that taken together can be drawn on a 2-manifold offset surface of that solid that will not divide the surface into two or more pieces.*

A *nonmanifold handle*, denoted g', is a "pinched" handle. That is, it is connected through one or more vertices or edges that have nonmanifold neighborhoods. Three tetrahedra may be combined to form a nonmanifold handle as is illustrated in Figure 2.9. The resulting solid has one

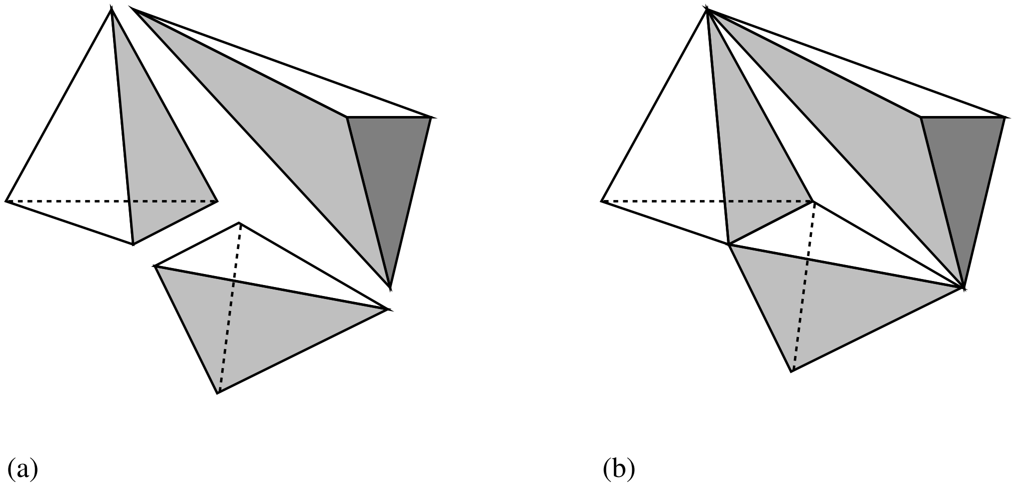


Figure 2.9: A solid with a nonmanifold handle.

nonmanifold handle that is pinched in three places. A (manifold) handle has no vertices or edges with nonmanifold neighborhoods.

This then gives us the ability to define nonmanifold handles and the genus of a nonmanifold solid.

Definition 2.19 *The number of nonmanifold handles, denoted g', is $g' = h'_1/2 - h_1/2$.*

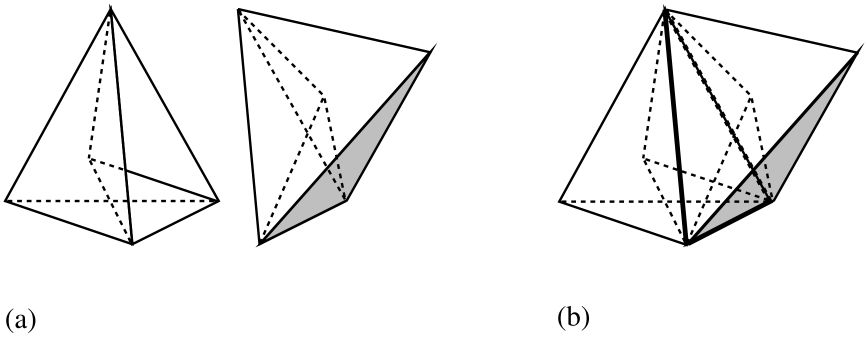


Figure 2.10: A solid with a nonmanifold chamber.

The last nonmanifold element that appears in our equations is the *nonmanifold chamber*. This is a bubble or space that is trapped when two surfaces (or two portions of the same surface) are joined together at a loop of edges with nonmanifold neighborhoods. For example, a nonmanifold chamber is is formed by joining the two modified tetrahedra of Figure 2.10. The loop of edges divides the surface of the solid into two portions: an inside surface of the solid (forming the chamber), and an outside surface of the solid. (Topologically speaking, the portion of the surface considered as the chamber is arbitrary.) We denote the number of nonmanifold chambers of a solid by c.

Finally, we are ready to explain the relationship between these elements of nonmanifold solids.

Theorem 2.1 *The Euler equation for regular solids is*

$$v' - e' + f = 2(s' - g)$$

where v', e', and s' are the numbers of vertex uses, edge uses, and shell uses, respectively.

Proof. We will systematically decompose the nonmanifold solid into its 2-manifold components. We begin by decomposing the edges with nonmanifold neighborhoods. For each edge with a nonmanifold neighborhood, we split the edge, introducing a new edge adjacent to the two vertices of the original edge and to one pair of adjacent faces. This will remove one edge use from the original edge, and will create a new edge with a single edge use. The total number of edge uses is unchanged. We repeat this process until no more edges have multiple edge uses, and $e' = e$.

Next, we decompose the vertices with nonmanifold neighborhoods. We split a vertex with a nonmanifold neighborhood and introduce a new vertex adjacent to one of the cycles of edges and faces. This will remove one vertex use from the original vertex, and will create a new vertex with a single vertex use. The total number of vertex uses does not change. We will have one of three cases:

1. If the original vertex and the new vertex are on the same shell use, we remove a nonmanifold handle. This does not affect the number of shells or shell uses.
2. If the original vertex and the new vertex are on different shell uses, but remain on the same shell, then we remove a nonmanifold handle. This does not affect the number of shells or shell uses.
3. If the original vertex and the new vertex are on different shell uses and separated sets of connected shell uses, then we split the shell. This partitions the shell uses into two separate shells, each with connected shell uses, and creating a new shell. The total number of shell uses is unchanged.

We repeat this process until no more vertices have multiple vertex uses, and $v' = v$. By our process of creating new shells, we will also have $s' = s$.

Once we have completed this process, the solid is decomposed into 2-manifold components and $v' = v$, $e' = e$, and $s' = s$. For these 2-manifold components, we know that

$$v - e + f = 2(s - g).$$

Therefore,

$$v' - e' + f = 2(s' - g).\square$$

Theorem 2.2 *The number of nonmanifold handles is determined by*

$$(v' - v) - (e' - e) - (s' - s) = g' - c.$$

Proof. Following our previous proof, we can confirm the relation of nonmanifold handles and chambers to other elements as we decompose the solid. First, we decompose the edges with nonmanifold neighborhoods. For each edge with a nonmanifold neighborhood, we split the edge, introducing a new edge adjacent to the two vertices of the original edge and to one pair of adjacent faces. This will remove one edge use from the original edge, and will create a new edge with a single edge use, thus $e' - e$ will be reduced by one. In addition, we will have one of two cases:

1. Splitting the edge removes a nonmanifold edge that is part of a cycle of nonmanifold edges, thereby destroying a nonmanifold chamber. Both c and $e' - e$ are reduced by one, and g' is unchanged.
2. Splitting the edge creates a nonmanifold handle. Then, $e' - e$ is reduced by one, and and g' is increased by one.

We repeat this process until no more edges have multiple edge uses. Therefore, $e' - e = 0$ and $c = 0$.

We next decompose the vertices with nonmanifold neighborhoods. We split a vertex with a nonmanifold neighborhood and introduce a new vertex adjacent to one of the cycles of edges and faces. This will remove one vertex use from the original vertex, and will create a new vertex with a single vertex use. The total number of vertex uses remains invariant. We will have one of three cases:

1. If the original vertex and the new vertex are on the same shell use, then the number of nonmanifold handles will be reduced by one. Therefore, the additional vertex balances the removed nonmanifold handle.
2. If the original vertex and the new vertex are on different shell uses, but remain on the same shell, then the number of nonmanifold handles will be reduced by one. Therefore, the additional vertex balances the removed nonmanifold handle.
3. If the original vertex and the new vertex are on different shell uses and different sets of connected shell uses, then we split the shell. This partitions the shell uses into two separate shells, each with connected shell uses, and creating a new shell. Therefore, the additional vertex cancels the additional shell.

We repeat this process until no more vertices have multiple vertex uses, and the solid is decomposed into 2-manifold components. Therefore, $v' - v = 0$, and $s' - s = 0$. Since we know that 2-manifold solids have no nonmanifold handles, it is clear that that

$$(v' - v) - (e' - e) - (s' - s) = g' - c.$$

The above operations are invertible. Thus the equation is preserved when applying these operations in either the forward or inverse directions. Using the 2-manifold components as the base case, we can construct any nonmanifold solid from its 2-manifold components. Therefore, by induction, we know that our equation holds for all nonmanifold solids.□

Examples

We can now see that our solid in Figure 2.4 satisfies this relation, since

$$v' - e' + f = 2(s' - g)$$

$$12 - 18 + 12 = 2(3 - 0).$$

$$(v' - v) - (e' - e) - (s' - s) = g' - c$$

$$(12 - 9) - (18 - 17) - (3 - 1) = 0 - 0$$

We can also verify some additional examples.

The solid in Figure 2.9(b) is composed of three tetrahedra. The first two tetrahedra are joined at two vertices and an edge, and the third is joined to the first two at two vertices. This solid has four vertices with nonmanifold neighborhoods and one edge with a nonmanifold neighborhood.

$$v' - e' + f = 2(s' - g)$$

$$12 - 18 + 12 = 2(3 - 0)$$

$$(v' - v) - (e' - e) - (s' - s) = g' - c$$

$$(12 - 8) - (18 - 17) - (3 - 1) = 1 - 0.$$

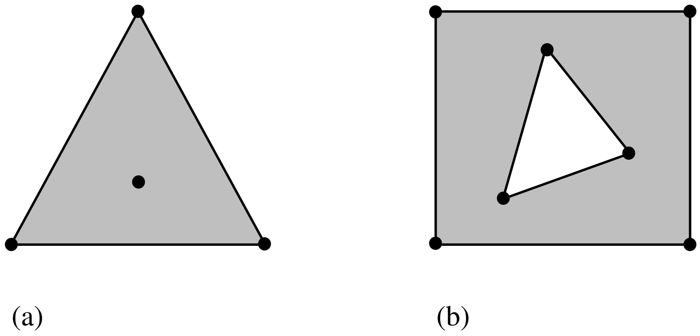
The solid in Figure 2.10(b) is composed of two manifold solids joined at three vertices and three edges. This solid has three vertices with nonmanifold neighborhoods. It also has three edges with nonmanifold neighborhoods in a cycle, forming a nonmanifold chamber.

$$v' - e' + f = 2(s' - g)$$

$$10 - 18 + 12 = 2(2 - 0)$$

$$(v' - v) - (e' - e) - (s' - s) = g' - c$$

$$(10 - 7) - (18 - 15) - (2 - 1) = 0 - 1.$$



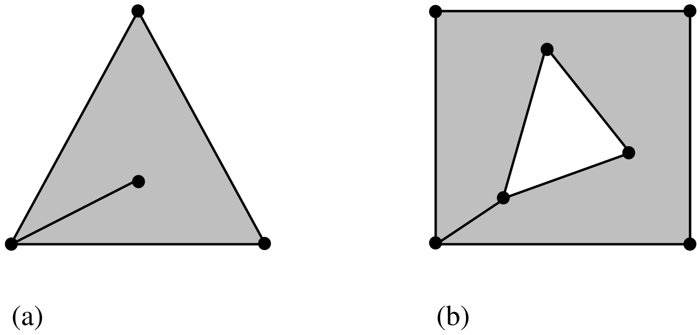
(a) (b)

Figure 2.11: Faces with multiple loops.

Multiple Face Loops

The Euler-Poincaré formula developed is restricted to faces represented as simple polygons. That is, a face may have only one boundary. We would like to represent faces with multiple boundaries, as shown in Figure 2.11.

Allowing faces with multiple boundaries requires an additional modification of the Euler-Poincaré formula.



(a) (b)

Figure 2.12: Simple faces.

A modification of the Euler-Poincaré formula was introduced by Braid, Hillyard, and Stroud [4] to additional face boundaries, which they denoted *rings*. Edges may be added to a face representation in order to connect the face boundaries. These edges are known as *artifact* or *bridge* edges. Artifact edges are added to the faces of Figure 2.11 as shown in Figure 2.12. The modified form of the Euler-Poincaré formula adds an artifact edge for each ring, denoted r,

$$v - (e + r) + f = 2(s - g).$$

We can allow faces with multiple boundaries in the generalized Euler-Poincaré formula by adding artifact edges. This extends the generalized Euler-Poincaré formula in a straightforward manner,

$$v' - (e' + r) + f = 2(s' - g).$$

We should also consider how multiple face loops affect the second equation. An artifact edge will always have exactly one edge use. Thus the equation

$$(v' - v) - (e' - e) - (s' - s) = g' - c$$

is unaffected by the addition of multiple face boundaries.

It is sometimes convenient to consider the total numbers of face boundaries, or *loops*, instead of the additional face boundaries, or rings. We can express the relationship between these three quantities,

$$r = (l - f),$$

where r, l, and f are the numbers of rings, loops, and faces.

Chapter 3

Generalized Split-Edge Representation

In this chapter, we will discuss existing boundary representations of solids, and discuss their advantages and limitations. A new boundary representation is introduced that allows representation of the desired domain of solids and is efficient for the given application.

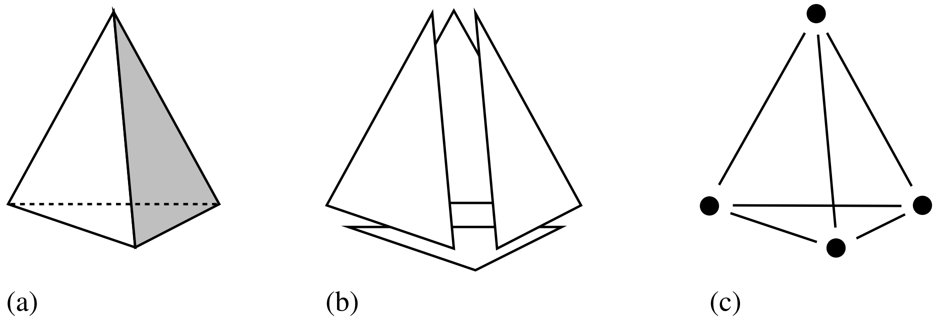


Figure 3.1: Elements of the boundary of a solid.

Solids may be represented by their bounding elements. For example, the tetrahedron in Figure 3.1(a) may be represented by its bounding faces (Figure 3.1(b)), its edges (Figure 3.1(c)), and its vertices (Figure 3.1(c)). In addition to these elements, we need to know the relations which describe the adjacencies of these elements, e.g. which faces are adjacent. The elements of the boundary and their adjacencies are together considered the topology. We also must know the dimensions and location of these elements, or geometry of the solids.

Since there are many ways to describe both the topology and geometry of solids, the choice of topological elements, the specific set of adjacency relations, and representation of geometry is critical to the amount of space needed to represent a solid, and the time needed to access information about the solid. For example, the choice of adjacency relations that are used depends on a number of factors: the set should be sufficient to fully determine all topological adjacencies; adjacency relations that are used most often should be directly accessible for efficiency; and redundant relations should not be included, except as required for access efficiency.

An extensive analysis of topological representations for manifold solids was conducted by Weiler [57]. Representations for nonmanifold solids have been introduced by Karasick [25], and others. These representations are discussed in this chapter, and a new representation for nonmanifold solids is introduced.

3.1 Winged-Edge Data Structure

Edge	next_face	Next face of this edge
	next_vertex	Next vertex of this edge
	next_cw_eh	Next clockwise edge
	next_ccw_eh	Next counterclockwise edge
	prev_face	Previous face of this edge
	prev_vertex	Previous vertex of this edge
	prev_cw_eh	Previous clockwise edge
	prev_ccw_eh	Previous counterclockwise edge

Table 3.1: The winged-edge data structure.

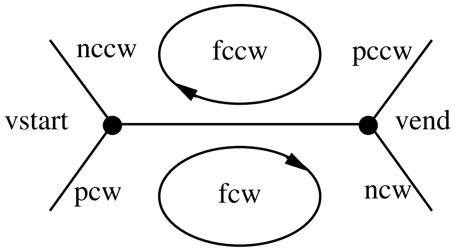


Figure 3.2: Edge relations in the winged-edge data structure.

The first explicit boundary representation of polyhedra was the winged-edge data structure by Baumgart [1]. It uses a record for each edge that contains the adjacencies between that edge and its two adjoining faces, and the preceding and succeeding vertices, and the preceding and succeeding edges in each of the faces.

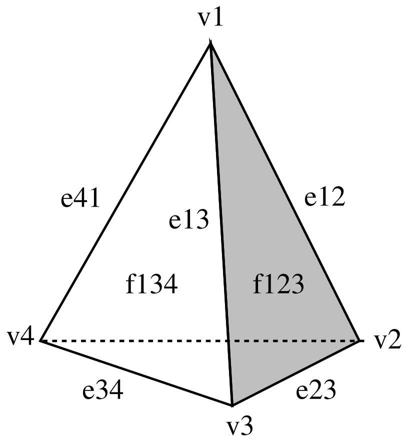


Figure 3.3: A simple solid.

edge	vstart	vend	pcw	ncw	pccw	nccw	fcw	fccw
e12	v1	v2	e13	e23	e24	e14	f123	f124
e13	v1	v3	e14	e34	e23	e12	f134	f123
e14	v1	v4	e12	e24	e34	e13	f124	f134
e23	v2	v3	e12	e13	e34	e24	f123	f234
e24	v2	v4	e23	e34	e14	e12	f234	f124
e34	v3	v4	e13	e14	e24	e23	f134	f234

Table 3.2: The winged-edge data structure for a tetrahedron.

Figure 3.2 shows the edge relations in the the winged-edge data structure. As an example, the tetrahedron in Figure 3.3 is represented using the winged-edge data structure in Table 3.2.

3.2 Split-Edge Data Structure

Edge-Half	edgeh_l	The parent loop
	cw_eh	Next clockwise edge-half
	ccw_eh	Next counterclockwise edge-half
	other_eh	Other edge-half
	edgeh_v	Vertex of this edge-half

Table 3.3: The split-edge data structure.

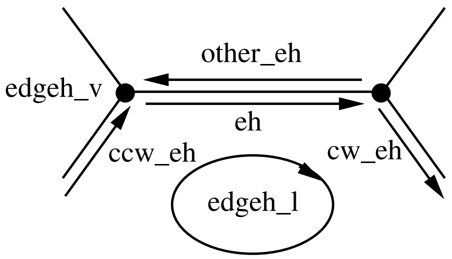


Figure 3.4: Edge relations in the split-edge data structure.

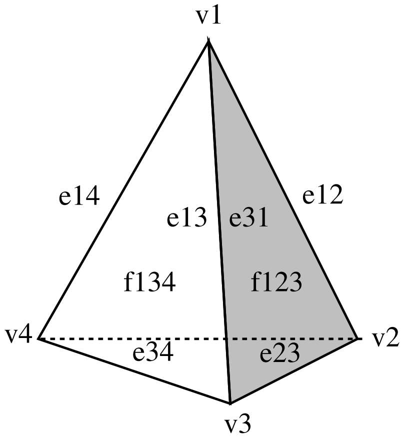


Figure 3.5: A simple solid.

The split-edge data structure is a variation of the winged-edge structure. It differs in that each edge is separated into two *edge-half* structures. One face and one vertex is associated with each edge-half, and each edge-half is associated with its other half.

Figure 3.4 shows the edge relations in the the winged-edge data structure. As an example, the tetrahedron in Figure 3.5 is represented using the winged-edge data structure in Table 3.4.

We would like a data structure that allows representation of nonmanifold as well as manifold solids. This is desirable since manifolds are not closed under the Boolean operations (or unary shape operations, as we will see).

Nonmanifold solid representations have been introduced, including Karasick's star-edge [25], Vanecek's fedge [56], and Laidlaw's representation [29]. The star-edge and fedge representations allow the more general definitions of faces we desire, but are not explicit (enough) to allow quick access from any element in a model to any other. This was not necessary when used in CSG modeling; however, is of utmost importance for matching and application of rules in boundary solid grammars. Laidlaw's representation is limited to triangulated faces. The next section introduces the generalized split-edge representation.

edge-half	edgeh_v	cw_eh	ccw_eh	other_eh	edgeh_l
eh12	v1	eh23	eh31	eh21	l123
eh21	v2	eh42	eh14	eh12	l124
eh13	v1	eh34	eh41	eh31	l134
eh31	v3	eh23	eh12	eh13	l123
eh14	v1	eh42	eh21	eh41	l124
eh41	v4	eh13	eh34	eh14	l134
eh23	v2	eh31	eh12	eh32	l123
eh32	v3	eh24	eh43	eh23	l234
eh24	v2	eh34	eh32	eh42	l234
eh42	v4	eh21	eh14	eh24	l124
eh34	v3	eh41	eh13	eh43	l134
eh43	v4	eh32	eh24	eh34	l234

Table 3.4: The split-edge data structure for a tetrahedron.

3.3 Generalized Split-Edge Representation

The generalized split-edge representation is an extension of the split-edge data structure. The relationships of the generalized split-edge representation are summarized in Table 3.5.

A vertex-use structure in the generalized split-edge representation is equivalent to a vertex use, as introduced in Chapter 2. A vertex of a solid is then represented by one or more vertex-uses. These vertex-uses are related by next_vertex_use relations. Each vertex of a manifold solid has exactly one vertex-use.

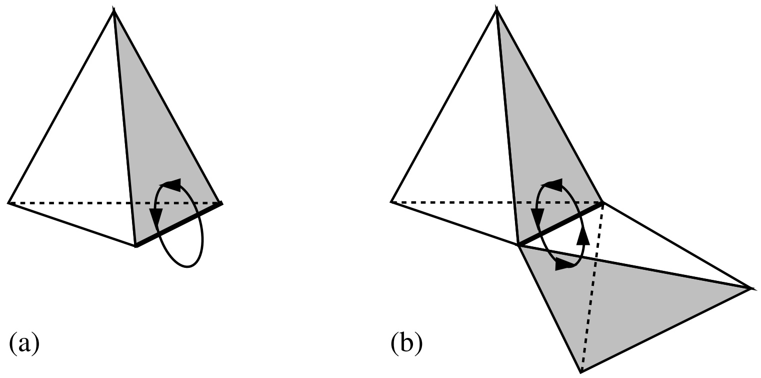


Figure 3.6: The Other_eh relations of edge-halves about two edges.

An edge use, as introduced in Chapter 2, is represented by exactly two edge-halves structures in the generalized split-edge representation, one for each face associated with the edge-use. An

Solid	next_solid	Next solid (in the world)
	prev_solid	Previous solid (in the world)
	solid_sh	First shell (use) in the solid
Shell-Use	shell_solid	The parent solid
	outside_sh	Next outside shell (use) in the solid
	connected_sh	Next connected shell-use of this shell (nonmanifold)
	inside_sh	First shell (use) inside this shell (use)
	shell_f	First face in this shell (use)
Face	face_sh	The parent shell (use)
	next_shell_f	Next face in the shell (use)
	prev_shell_f	Previous face in the shell (use)
	face_l	First loop in the face
Loop	loop_f	The parent face
	next_face_l	Next loop in the face
	loop_eh	First edge in loop
	loop_v	Single vertex (use) in loop (only in edgeless loop)
Edge-Half	edgeh_l	The parent loop
	cw_eh	Next clockwise edge-half
	ccw_eh	Next counterclockwise edge-half
	other_eh	Other edge-half (for nonmanifold - next in cycle)
	edgeh_v	Vertex (use) of this edge-half
Vertex-Use	vertex_l	The parent loop (in edgeless loop)
	vertex_eh	First edge-half of this vertex-use
	next_vertex_use	Next vertex-use of the vertex

Table 3.5: The generalized split-edge representation.

edge of a solid is then represented by a pair of edge-halves for each of its edge-uses. The other_eh relation denotes the adjacencies of the edge uses of an edge. Since each edge of a manifold solid

edge-half	edgeh_v	cw_eh	ccw_eh	other_eh	edgeh_l
eh12	v1	eh23	eh31	eh21	l123
eh21	v2	eh42	eh14	eh12	l124
eh13	v1	eh34	eh41	eh31	l134
eh31	v3	eh23	eh12	eh13	l123
eh14	v1	eh42	eh21	eh41	l124
eh41	v4	eh13	eh34	eh14	l134
eh23	v2	eh31	eh12	eh32	l123
eh32	v3	eh24	eh43	eh23	l234
eh24	v2	eh34	eh32	eh42	l234
eh42	v4	eh21	eh14	eh24	l124
eh34	v3	eh41	eh13	eh43	l134
eh43	v4	eh32	eh24	eh34	l234

Table 3.6: The generalized split-edge representation for a tetrahedron.

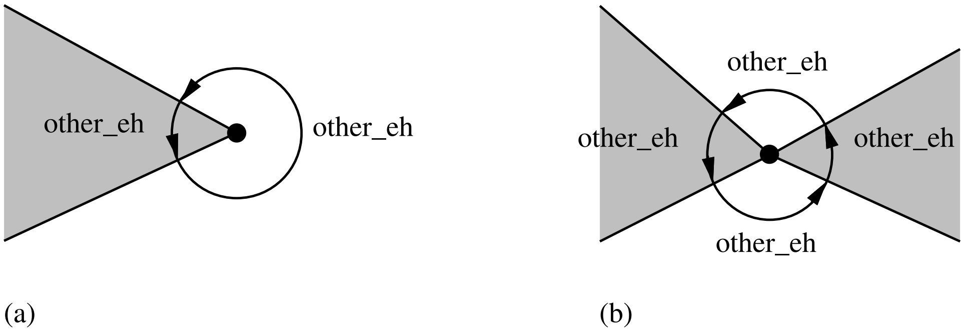


Figure 3.7: A cross section through two edges.

has a single edge-use, it is represented by exactly two edge-halves.

Shells are ordered in a tree by containment. Shells that are connected by one or more vertices or edges with nonmanifold neighborhoods are linked using connected_sh relations. The shell uses of the generalized split-edge representation correspond to the shell uses described in Chapter 3.

Chapter 4

Local Operations on Solids

Local modifications to solids are accomplished using *local operations.* Euler operations modify a solid's topology. Geometric operations modify the geometric description of a solid. This chapter describes these local operations, with the primary emphasis on manifold and nonmanifold Euler operations.

4.1 Euler Operations

Euler operations are a set of operators that manipulate graph representations of the topological elements and adjacencies of the boundary of solids. They modify a boundary representation by adding and removing topological elements, while maintaining consistent topological adjacency relations. Typical Euler operations split edges (Figure 4.1), and split faces (Figure 4.2).

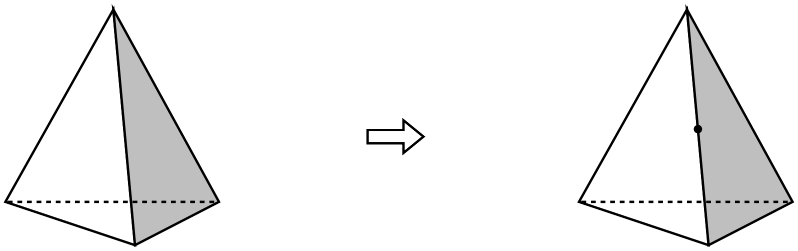


Figure 4.1: Splitting an edge.

Euler operations were introduced by Baumgart [2] for use on his winged-edge polyhedron representation. They have since been adopted by several research groups [13, 4, 22, 33]. The topological validity and completeness of Euler operations has been presented by Eastman and Weiler [14], and demonstrated rigorously by Mäntylä [34]. Fitzhorn demonstrated that Euler operators may be represented as graph grammar productions [15].

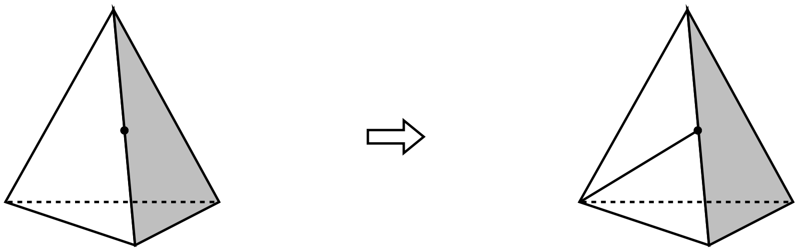


Figure 4.2: Splitting a face.

The Euler operators in this thesis follow the naming convention originally introduced by Baumgart. The names describe the effect the operators have on the creation and removal of topological elements as well as the genus of the solid. An `m` stands for "make" or create, and `k` stands for "kill" or remove. Each of these is followed by the letters signifying the types of topological elements created or removed. `v, e, l, f, s` and `g` stand for vertex, edge, loop, face, shell, and genus. Thus, `mev` stands for "make edge, vertex", and `keml` stands for "kill edge, make loop". A few operators, such as `glue` and `esplit`, have more descriptive names.

Manifold Euler Operations

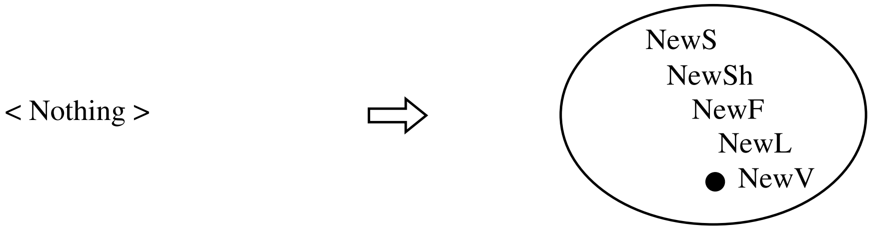


Figure 4.3: The mssflv operator

`mssflv(NewS,NewSh,NewF,NewL,NewV)`
: "make solid, shell, face, loop, vertex" creates a new solid topology with a single shell (manifold surface) with one face (containing a single loop) and one vertex. No edges are created. This is the minimal topology needed to represent a shell, but is not sufficient to represent a polyhedron. The identifiers of the new elements are returned as the values of "NewS", "NewSh", "NewF", "NewL", and "NewV". The `mssflv` operator is illustrated in Figure 4.3.

`merge_solids(S1,S2)`
: merges the shells of two solids. The shells of "S2" are added to the solid "S1", and the record of "S2" is removed.

`msflv(S,NewSh,NewF,NewL,NewV)`
: "make shell, face, loop, vertex" adds a new shell (manifold surface) to an existing solid. The

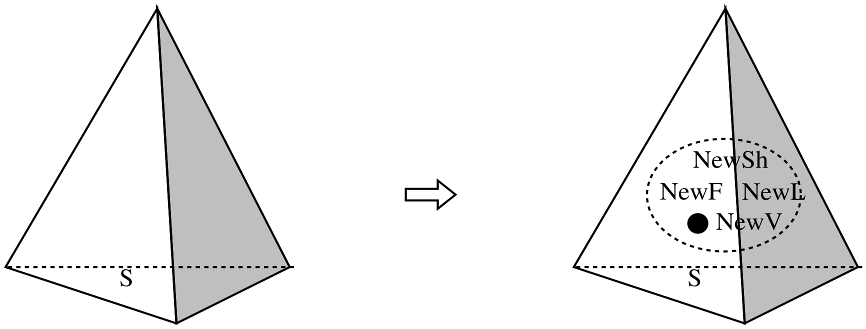


Figure 4.4: The msflv operator

new shell “NewSh” is added to the given solid “S”. The new shell, face, loop and vertex are returned as the values of “NewSh”, “NewF”, “NewL”, and “NewV”. As in **`mssflv`**, the new shell consists of a single face, loop and vertex. The **`msflv`** operator is illustrated in Figure 4.4.

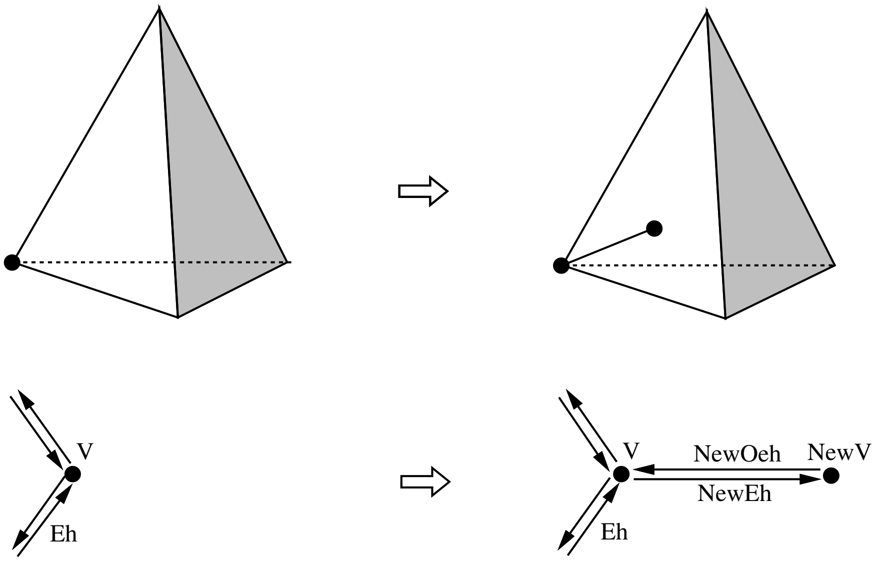


Figure 4.5: The mev operator

`mev(V,Eh,NewV,NewEh)`

“make edge, vertex” creates a new edge and vertex as a strut edge in a face (see Figure 4.5). For a shell with no edges, **`mev`** creates the first edge in the shell’s single face and loop. Both sides of the new edge will be adjacent to the same face. “V” is the existing vertex which will be at one end of the new edge. “Eh” is the edge-half that is counterclockwise from the edge-half of vertex “V”. If there are no edges adjacent to vertex “V”, “Eh” will be null. When **`mev`** is complete:

- the new edge-half "NewEh" will be clockwise to the given edge-half "Eh";
- the new vertex "NewV" will be the vertex of the other_eh and the cw_eh of the new edge-half "NewEh".

The operation of `mev` is illustrated in Figure 4.5.

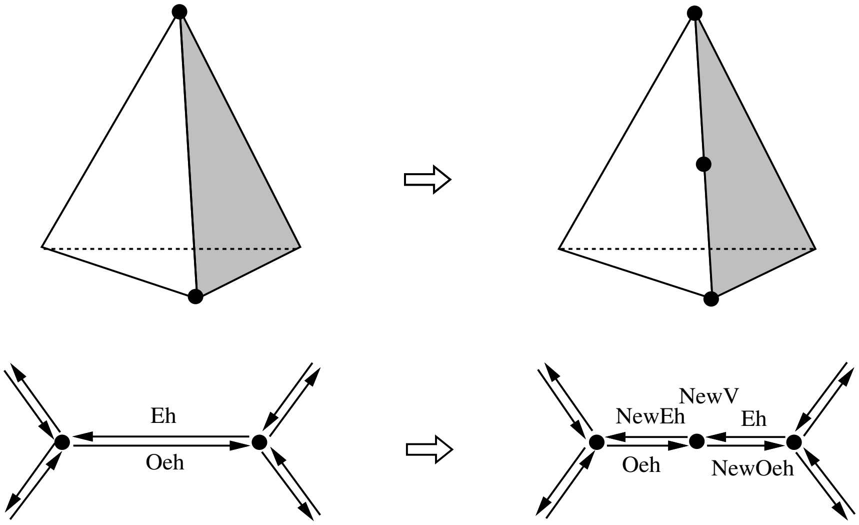


Figure 4.6: The esplit operator

`esplit(Eh,NewEh,NewV)`

"edge split" splits a given edge "Eh", creating a new edge and vertex. `esplit` is a form of `mev`.

When `esplit` is complete:

- the new edge-half "NewEh" will be clockwise to the given edge-half "Eh";
- the second new edge-half will be the other_eh of the given edge-half "Eh";
- the new vertex "NewV" will belong to both the two new edge-halves.

The `esplit` operator is illustrated in Figure 4.6.

`mefl(V1,PredEh,V2,SuccEh,NewEh,NewL,NewF)`

"make edge, face, loop" splits a face and loop, creating a new face, loop, and an edge separating the new and old faces. The vertex "V1" belongs to the clockwise edge-half of "PredEh". If the vertex "V1" has no edge-halves, "PredEh" will be null. The vertex "V2" belongs to "SuccEh". If the vertex "V2" has no edge-halves, "SuccEh" will be null. When `mefl` is complete:

- the new edge-half "NewEh" will connect the vertices "V1" and "V2", and will be clockwise to "PredEh", and counterclockwise to "SuccEh";

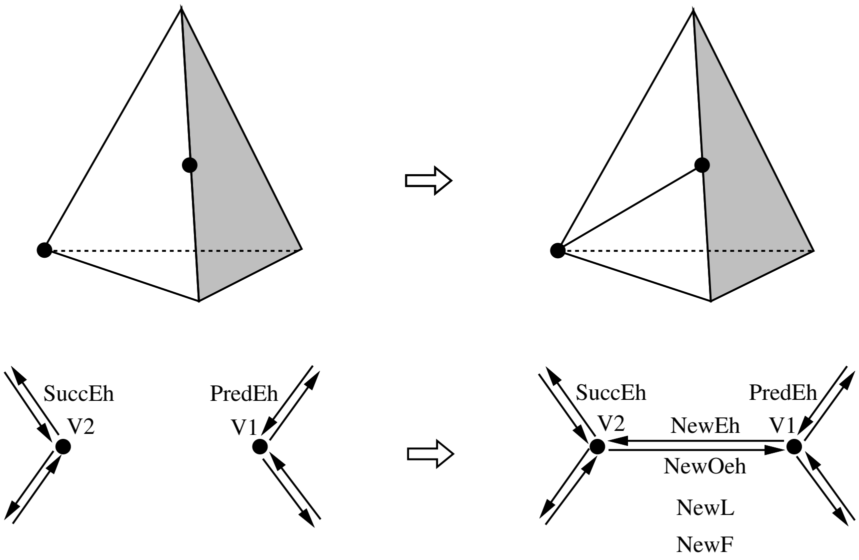


Figure 4.7: The mefl operator

- the second new edge-half will be the other_eh of "NewEh";
- the loop of the other_eh of "NewEh" will be the new loop "NewL";
- the face of the new loop "NewL" will be the new face "NewF".

If the vertices are the same ("V1" = "V2"), `mefl` creates an edge with the same vertex at each end. The use of the `mefl` operator is illustrated in Figure 4.7.

`keml(Eh,NewL)`
"kill edge, make loop" removes an edge on a face splitting one loop into two separate loops, creating a new loop of edges. If the edge is the only one in the loop, the result will be two loops, each containing a single vertex. If the edge is a strut edge, one of the loops will contain a single vertex. The edge-halves "Eh" and its other_eh will be removed. The counterclockwise edge-half of "Eh" (or the vertex of "Eh") will be contained in the new loop "NewL". The `keml` operator is illustrated in Figure 4.8.

`glue(F1,F2)`
"kill faces, loops" glues two faces together. The faces must have the same number of loops, and the corresponding loops must have the same number of edges and vertices. The use of the `glue` operator is illustrated in Figure 4.9.

Inverse Manifold Euler Operations

`kssflevs(S)`
"kill solid, shells, faces, loops, edges, vertices" deletes the existing solid "S" and all of its elements, as illustrated in Figure 4.10.

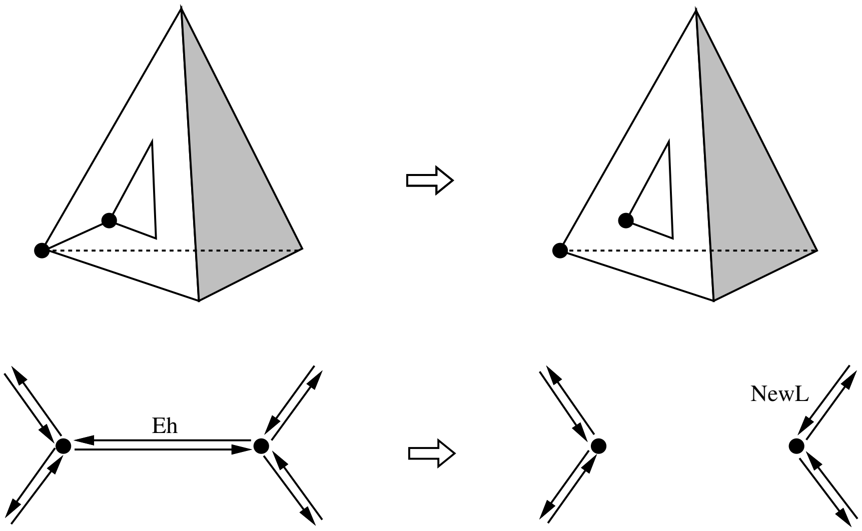


Figure 4.8: The keml operator

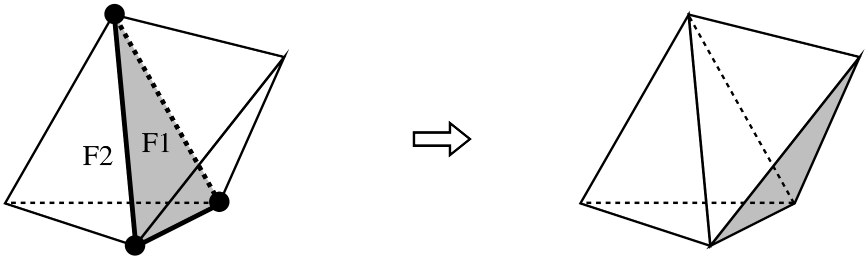


Figure 4.9: The glue operator

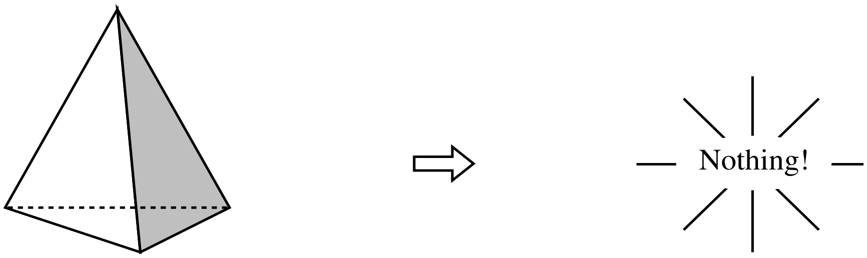


Figure 4.10: The kssflevs operator

`ksflevs(Sh)`

"kill shell, faces, loops, edges, vertices" Remove the shell "Sh" and all its elements from an existing solid. The **`ksflevs`** operator is illustrated in Figure 4.11.

`kev(Eh)`

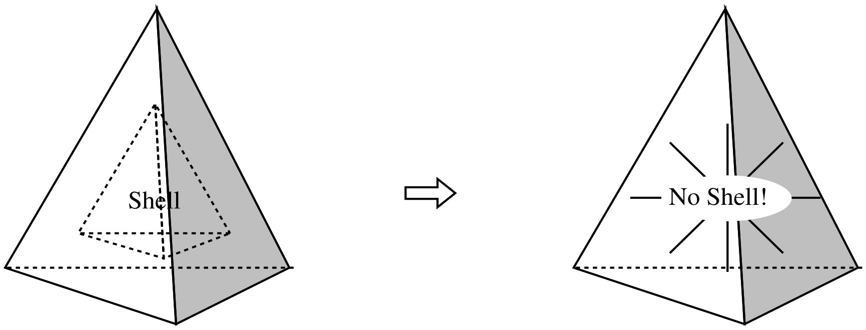


Figure 4.11: The ksflevs operator

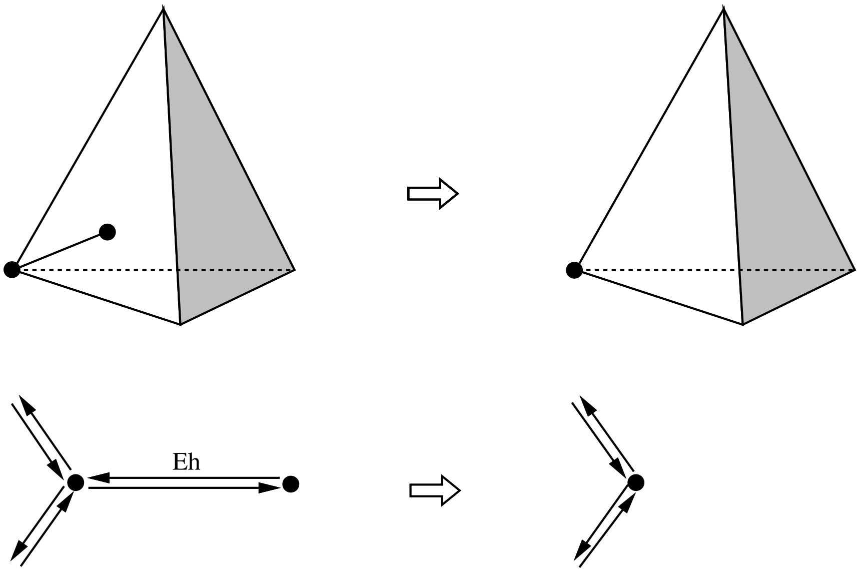


Figure 4.12: The kev operator

"kill edge, vertex" removes a "strut" edge from a face, and the vertex belonging to it. The use of the **kev** operator is illustrated in Figure 4.12.

ejoin(Eh)

"edge split" joins an edge, creating a new edge and vertex. **ejoin** is a form of **kev**. **ejoin** applies when the vertex of "Eh" is adjacent to only two edge-halves, "Eh" and the cw_eh of the other_eh of "Eh". When **ejoin** is complete, "Eh" and the cw_eh of the other_eh of "Eh" will be removed, as well as the vertex adjacent to these two edge-halves. The **ejoin** operator is illustrated in Figure 4.13.

esqueeze(Eh)

"edge squeeze" joins two vertices, removing the edge between. **esqueeze** is a form of **kev**. "Eh" and its other_eh will be removed, as well as the vertex belonging to "Eh". The other

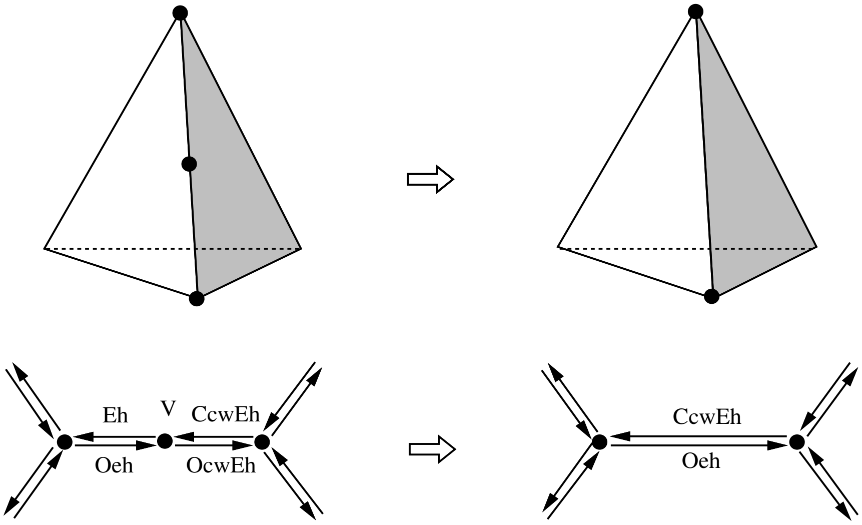


Figure 4.13: The ejoin operator

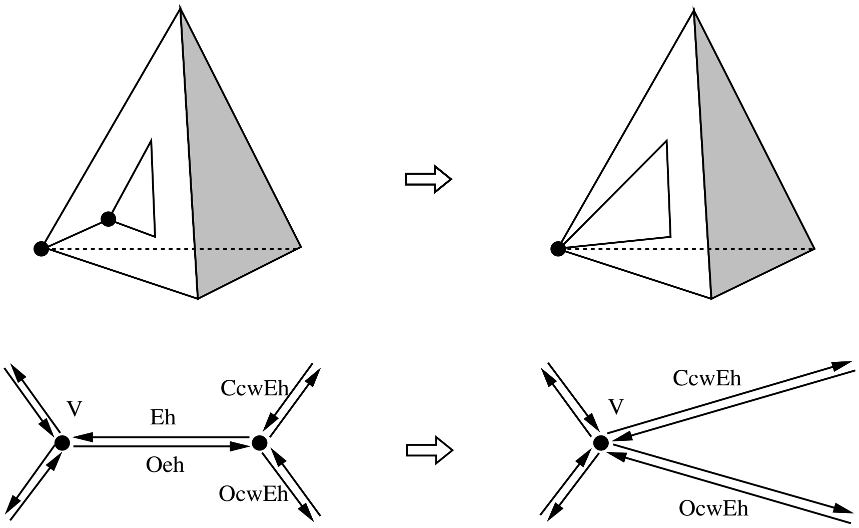


Figure 4.14: The esqueeze operator

edge-halves adjacent to the vertex of "Eh" will now be adjacent to the vertex that belonged to the other_eh of "Eh". The `esqueeze` operator is illustrated in Figure 4.14.

`kefl(Eh)`

"kill edge, face, loop" joins a face and loop, removing the edge separating the two faces, and the face and loop of that edge. The operation of `kefl` is illustrated in Figure 4.15.

`mekl(V1,PredEh,V2,SuccEh,NewEh)`

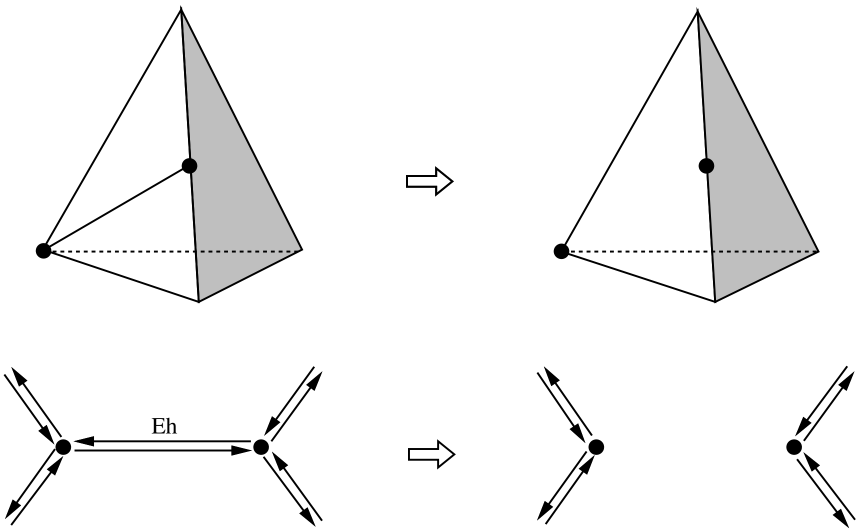


Figure 4.15: The kefl operator

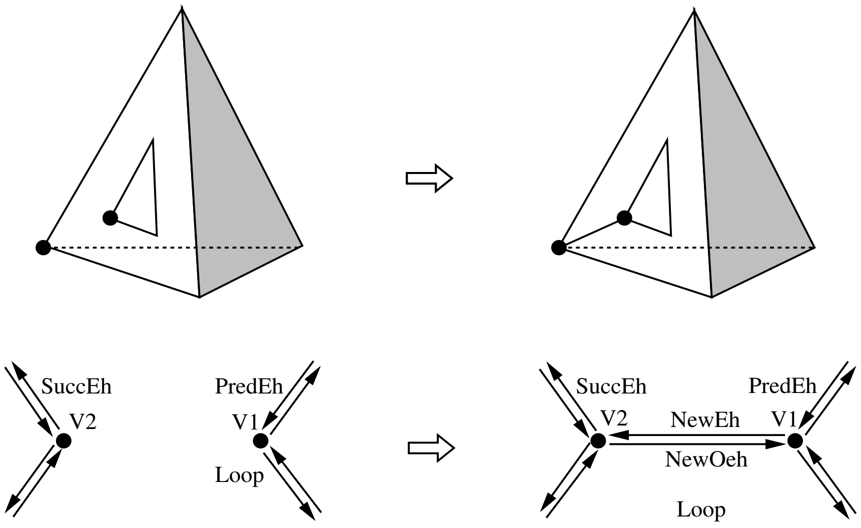


Figure 4.16: The mekl operator

“make edge, kill loop” adds an edge between two vertices, V1 and V2, on different loops of a face. The vertex “V1” belongs to the clockwise edge-half of “PredEh”. If the vertex “V1” has no edge-halves, “PredEh” will be null. The vertex “V2” belongs to “SuccEh”. If the vertex “V2” has no edge-halves, “SuccEh” will be null. When `mekl` is complete:

- the new edge-half “NewEh” will connect the vertices “V1” and “V2”, and will be clockwise to “PredEh”, and counterclockwise to “SuccEh”;
- the second new edge-half will be the other_eh of “NewEh”;

The `mekl` operator is illustrated in Figure 4.16.

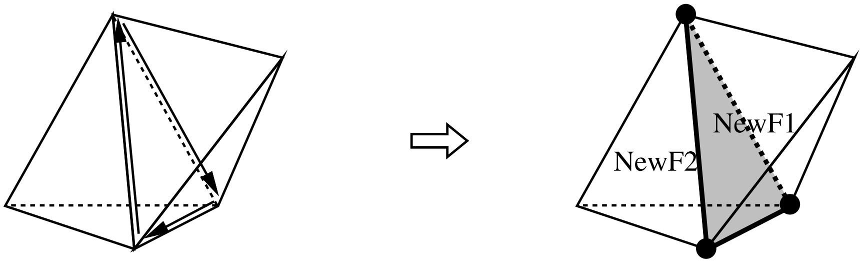


Figure 4.17: The unglue operator

`unglue(CycleOfEhs,NewF1,NewF2)`

"make faces, loops" unglues a cycle of edges, creating two new faces and loops. `CycleOfEhs` is a list of the pairs of edge-halves to be separated. The use of the `unglue` operator is illustrated in Figure 4.17.

Using Manifold Euler Operations

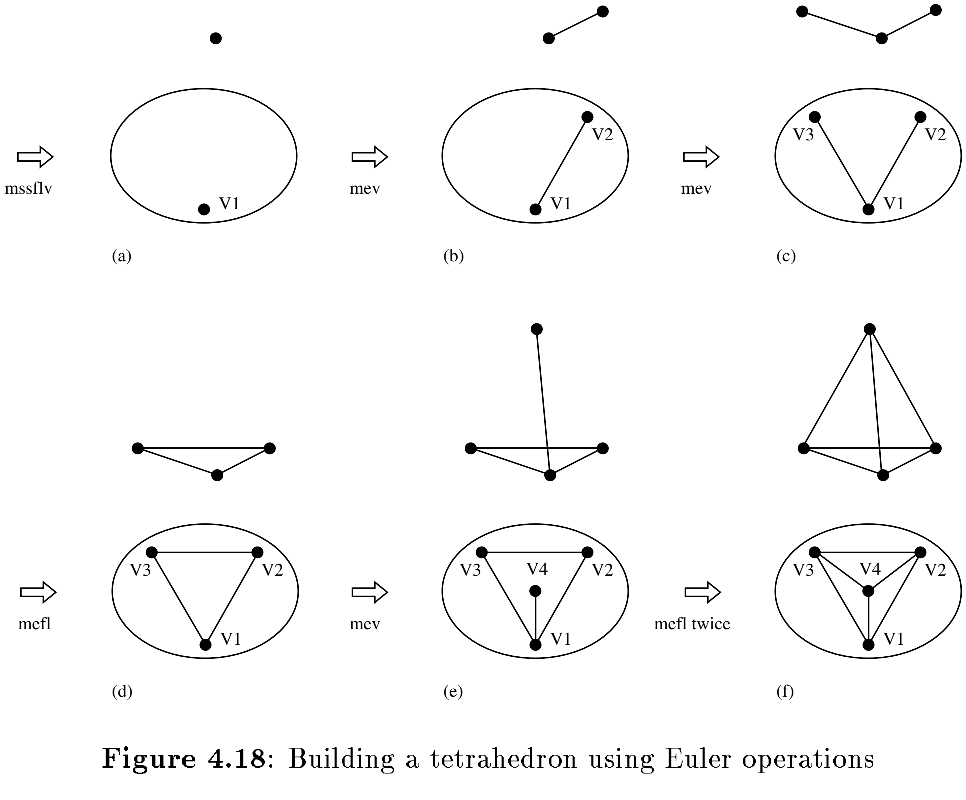


Figure 4.18: Building a tetrahedron using Euler operations

We can illustrate the use of Euler operators with a simple example, the construction of a tetrahedron. The steps of the construction are presented Figure 4.18, with both plane models and three-dimensional models.

The construction begins with a minimal topology, created with **`mssflv`** (a). The first and second edges are then added with two applications of **`mev`** (b and c). At this point, the model has a single face. By creating an edge between $V2$ and $V3$, we split the face using **`mefl`** (d). This gives us a triangular lamina. $V4$ and the strut edge between $V1$ and $V4$ are created with **`mev`** (e). Two more applications of **`mefl`** create the remaining two faces and two vertices, completing the topology of the tetrahedron (f). In order to complete the description of the tetrahedron, we need to assign coordinates to the vertices.

4.2 Nonmanifold Euler Operations

The Euler operators are complete and valid for manifold solids. We would like to extend the Euler operators to allow the manipulation of both manifold and nonmanifold solids. Requicha and Voelcker [44] conjectured that Euler operators could be extended to nonmanifold solids. This section discusses existing Euler operators on nonmanifold solids, and introduces new ones.

A set of generalized Euler operators for nonmanifold solids has previously been proposed by Desaulniers and Stewart [11]. Their method uses manifold Euler operators to modify the topology, while introducing infinitesimal edges and faces at points with nonmanifold neighborhoods. Specifically, a nonmanifold Euler operation would be composed of zero or more applications of **`mev`**, followed by one or more applications of **`mefl`**.

We introduce three nonmanifold Euler operations (and their inverses) to construct explicit representations of nonmanifold solids using the generalized split-edge data structure. These maintain the additional nonmanifold adjacency relationships: multiple loops of faces about a vertex; multiple pairs of faces about an edge; and and multiple connected uses (components) of a shell.

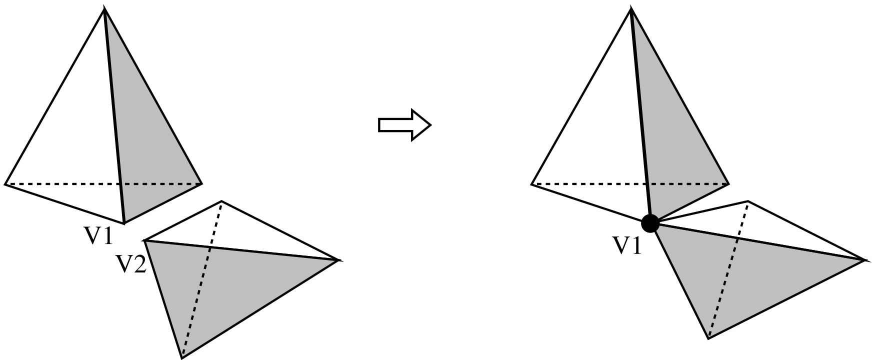


Figure 4.19: The ksv operator

ksv(V1,V2)

"kill shell, vertex" merges two vertices existing on different shells, and merges their shells. "V1" and "V2" are the two vertices to be merged. When **ksv** is complete, "V1" and "V2" will be uses of the same nonmanifold vertex, and will be on different uses of the same shell. The operation of **ksv** is illustrated in Figure 4.19.

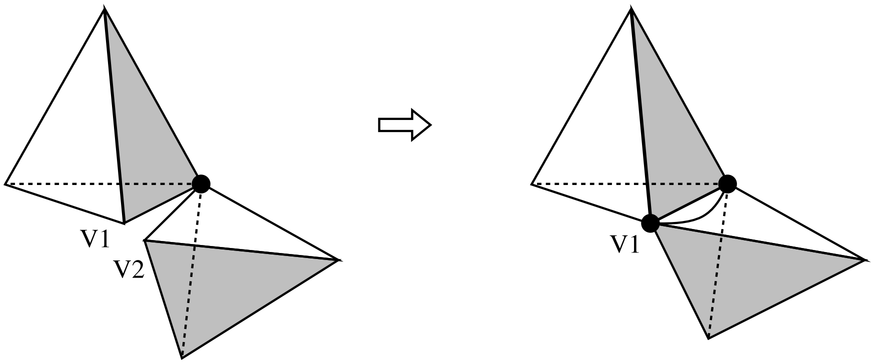


Figure 4.20: The kvmg operator

kvmg(V1,V2)

"kill vertex, make genus" merges two vertices existing on the same shell. This creates a (nonmanifold) handle and increases the genus by one. "V1" and "V2" are the two vertices to be merged. When **kvmg** is complete, "V1" and "V2" will be uses of the same nonmanifold vertex, and will be on the same shell and shell use. The **kvmg** operator is illustrated in Figure 4.20.

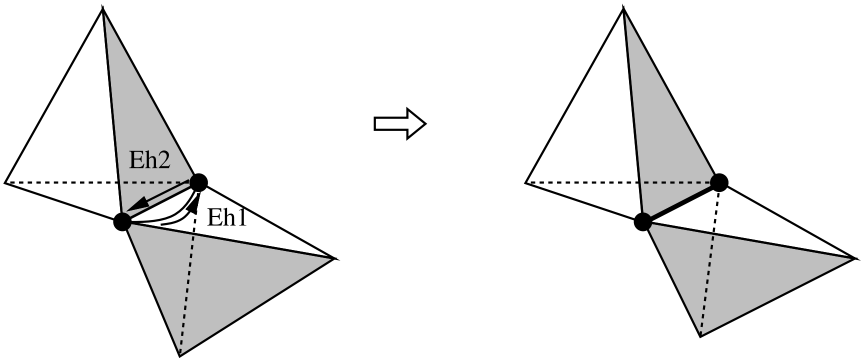


Figure 4.21: The keg operator

keg(Eh1,Eh2)

"kill edge, genus" merges two edges that share their two vertices. This seals a (nonmanifold) handle and decreases the genus by one, or creates a nonmanifold chamber. "Eh1" and "Eh2" are halves of the two edges to be merged. When **keg** is complete, “Eh1” and “Eh2” will be halves of the same nonmanifold edge. The use of the **keg** operator is illustrated in Figure 4.21.

Inverse Nonmanifold Euler Operations

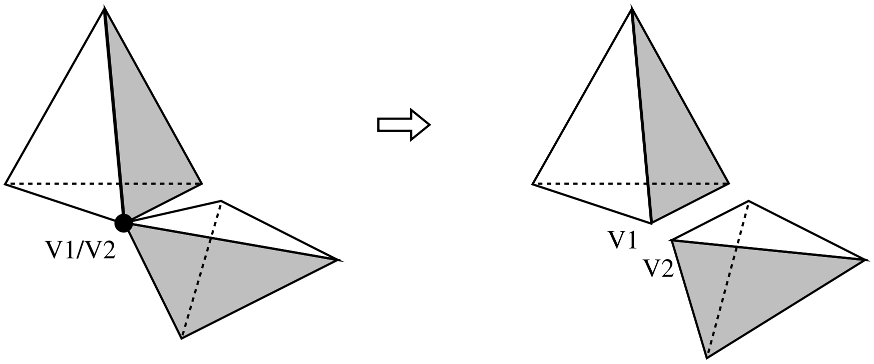


Figure 4.22: The msv operator

`msv(V1,V2)`

“‘make shell, vertex” splits two vertex-uses of a vertex. The two vertex-uses are on the same shell, but different shell-uses, and creates a shell. “V1” and “V2” are different uses of the same vertex that are to be separated. “V1” and “V2” must be the only vertex uses joining their shell uses. When **msv** is complete, “V2” will be removed from the vertex of “V1” and will form a separate vertex, and will be on a separate shell. The **msv** operator is illustrated in Figure 4.22.

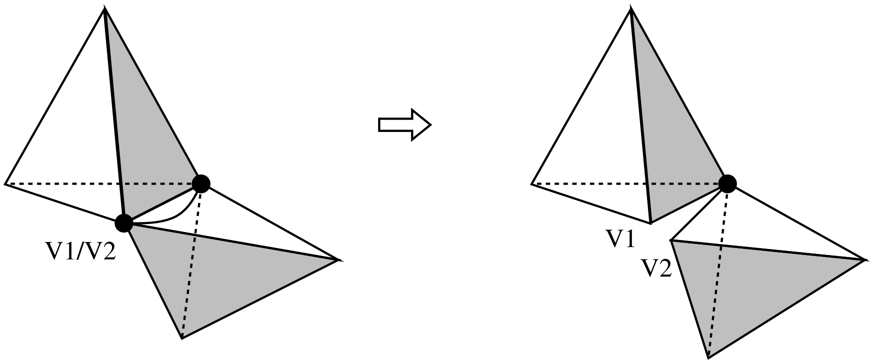


Figure 4.23: The mvkg operator

`mvkg(V1,V2)`

“make vertex, kill genus” splits two vertex-uses of a vertex that exist on the same shell and shell-use. This removes a (nonmanifold) handle and decreases the genus by one. The **mvkg** operator is illustrated in Figure 4.23.

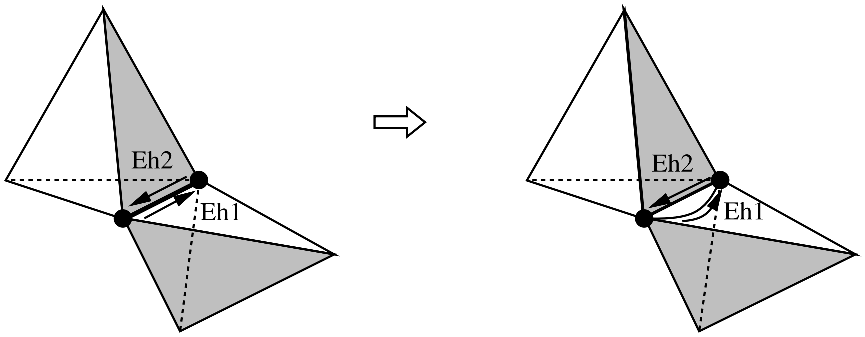


Figure 4.24: The meg operator

`meg(Eh1,Eh2)`

"make edge, genus" splits two edge-uses of an edge. This creates a (nonmanifold) handle and increases the genus by one, or removes a nonmanifold chamber. The **meg** operator is illustrated in Figure 4.24.

Using Nonmanifold Euler Operations

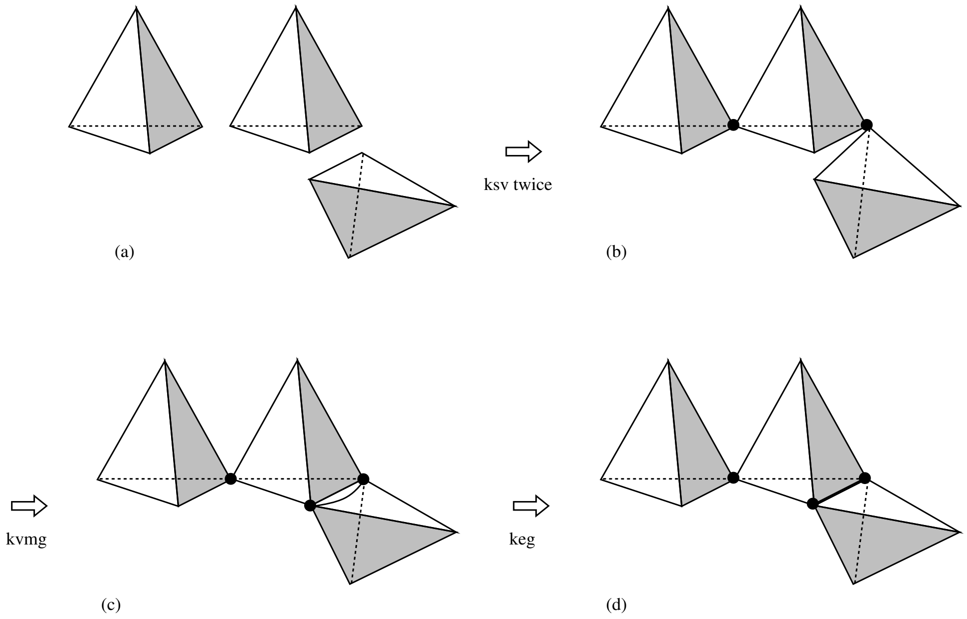


Figure 4.25: Building a solid using nonmanifold Euler operations

We can illustrate the use of nonmanifold Euler operators with another example, constructing the solid in Figure 2.4. The steps of the construction are presented Figure 4.25.

The construction begins with three tetrahedra (a). The three are connected with two applications of `ksv` (b). Two vertices are then joined with one application of `kvmg`, making a nonmanifold handle (c). One application of `keg` joins the two edges and removes the nonmanifold handle (d), and completes the solid.

Properties of Euler Operations

	v	e	l	f	s
`msflv`	$+1$	0	$+1$	$+1$	$+1$
`mev`	$+1$	$+1$	0	0	0
`mefl`	0	$+1$	$+1$	$+1$	0
`mekl`	0	$+1$	-1	0	0
`ksflevs`	$-v$	$-e$	$-l$	$-f$	-1
`kev`	-1	-1	0	0	0
`kefl`	0	-1	-1	-1	0
`keml`	0	-1	$+1$	0	0

Table 4.1: The effect of manifold Euler operators on the numbers of topological elements

	v'	e'	l	f	s'	h	h'	c
`glue`$_1$	$-v_c$	$-e_c$	-2	-2	-1	0	0	-1
`glue`$_2$	$-v_c$	$-e_c$	-2	-2	0	$+1$	0	-1
`unglue`$_1$	$+v_c$	$+e_c$	$+2$	$+2$	$+1$	0	0	$+1$
`unglue`$_2$	$+v_c$	$+e_c$	$+2$	$+2$	0	-1	0	$+1$

Table 4.2: The effect of the `glue` and `unglue` operators on the numbers of topological elements

The effect of the manifold Euler operators on the numbers of topological elements is presented in Table 4.1. Technically, `ksflevs` is a nonmanifold Euler operations since it removes all the elements with their manifold and nonmanifold adjacencies.

Both forms of `glue` remove the nonmanifold vertex uses v_c and edge uses e_c around the nonmanifold chamber. In addition, `glue`$_1$ presumes that the nonmanifold chamber separates two shell uses, and therefore removes a shell use. `glue`$_2$ presumes that the nonmanifold chamber separates two portions of the same shell use, and thus forms a (manifold) handles. The effect of Euler operators on the numbers of topological elements is presented in Table 4.2.

	v	e	s	h'	c
ksv	-1	0	-1	0	0
kvmg	-1	0	0	$+1$	0
keg$_1$	0	-1	0	-1	0
keg$_2$	0	-1	0	0	$+1$
msv	$+1$	0	$+1$	0	0
mvkg	$+1$	0	0	-1	0
meg$_1$	0	$+1$	0	$+1$	0
meg$_2$	0	$+1$	0	0	-1

Table 4.3: The effect of nonmanifold Euler operators on the numbers of topological elements

The effect of the nonmanifold Euler operators on the numbers of topological elements is presented in Table 4.3.

4.3 Geometric Operations

Modifications to the geometry of solids are implemented using geometric operations. For the purposes of this thesis, we limit the scope of our geometric operations to vertex coordinate assignment. From the vertex coordinates, we can derive edge equations and face equations.

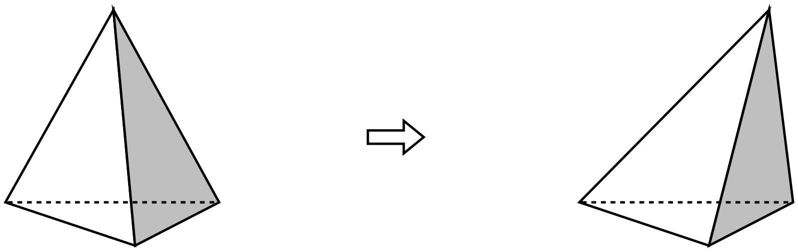


Figure 4.26: Moving a vertex.

The assignment of new coordinates to an existing vertex allows us to modify the description of a solid, as shown in Figure 4.26. The `set_vertex` operation is used for this purpose.

`set_vertex(V,[X,Y,Z])`
Set the coordinates of the given vertex to `[X, Y, Z]`.

Chapter 5

Unary Shape Operations

This chapter introduces the unary shape operations. In order to guarantee that we always have models that represent solid objects, we need an operation that takes any model with a valid topology and arbitrary (possibly invalid) geometry, and produces a valid model. The operation should leave a valid model unchanged. Changes introduced to an invalid model should be consistent with the given topology and geometry, while removing self-intersections of the surface. The *unary shape operations* are designed to serve this purpose.

Modeling with Euler operations allows us to modify models, while guaranteeing that the topology of a model is valid. This allows us much freedom in creating and modifying the surfaces of models. However, if we are allowed unrestricted assignment of geometry, we may construct models that have self-intersections and inconsistent face descriptions.

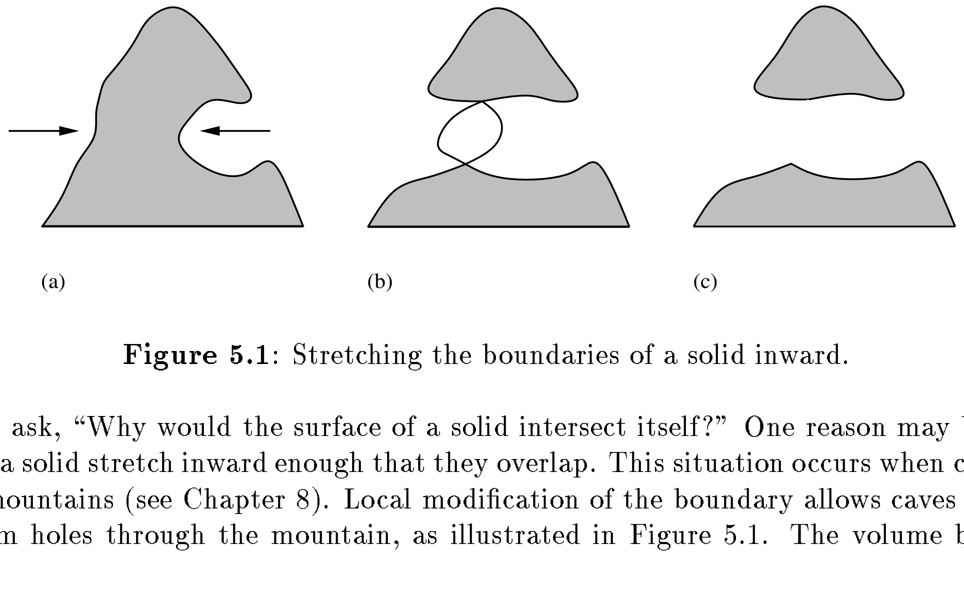


Figure 5.1: Stretching the boundaries of a solid inward.

We can ask, "Why would the surface of a solid intersect itself?" One reason may be that two surfaces of a solid stretch inward enough that they overlap. This situation occurs when constructing "fractal" mountains (see Chapter 8). Local modification of the boundary allows caves to intersect and to form holes through the mountain, as illustrated in Figure 5.1. The volume between the two intersecting parts of the boundary is to the outside of both. Since the boundary separates the outside from itself, we really have a hole through the solid. The intersecting portions of the boundary are unwanted.

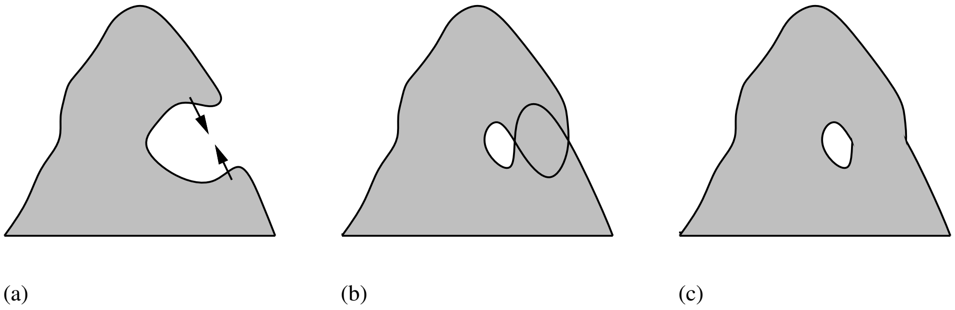


Figure 5.2: Stretching the boundaries of a solid outward.

Alternately, two parts of a solid may stretch outward far enough to intersect. On the example mountain, modification of the boundary creates spires that intersect back into the mountain, forming arches. We can see this in Figure 5.2. Then we have parts of the boundary inside what we think of as solid. Since the boundary separates interior points of the inside, the intersecting portion of the boundary is unnecessary. It is also undesirable, since computation of the volume based on the boundary would be incorrect.

The *unary union* of a solid (defined by its boundary) is the solid that contains all the points enclosed at least once by the boundary. For example, the unary union of the solid in Figure 5.1(b) is shown is Figure 5.1(c). The unary union of the solid in Figure 5.2(b) is shown is Figure 5.2(c). The *unary intersection* of a solid is the solid that contains all the points enclosed more than once by the boundary.

5.1 Winding Numbers

We can define the unary shape operations more precisely using *winding numbers.*

Definition 5.1 *The* **winding number** *of a point p with respect to an oriented 2-manifold surface $\mathcal{S}$ embedded in E^3 is the number of times the surface encloses the point, and is denoted $w(\mathcal{S}, p)$. The winding number is defined for any point not on the surface.*

Since 2-manifold surfaces are closed, we will always have integer winding numbers. The term *enclosing number* is more intuitive when considering points enclosed by closed surfaces [7]. We are also interested in representing solids with multiple bounding surfaces, so we need to expand our definition.

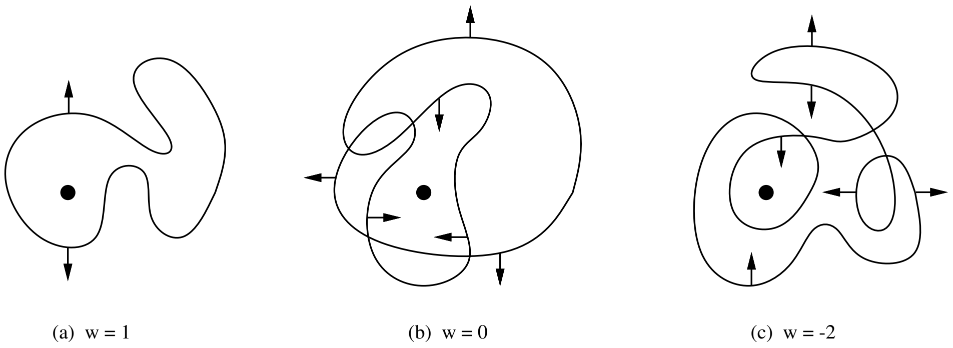


Figure 5.3: Winding numbers.

Definition 5.2 *The* **winding number** *of a point p with respect to multiple surfaces is the sum of the winding numbers of that point with respect to each of the individual surfaces,* $\sum_{i=1}^{n} w(\mathcal{S}_i, p)$. *The winding number is defined for any point not on any of the surfaces.*

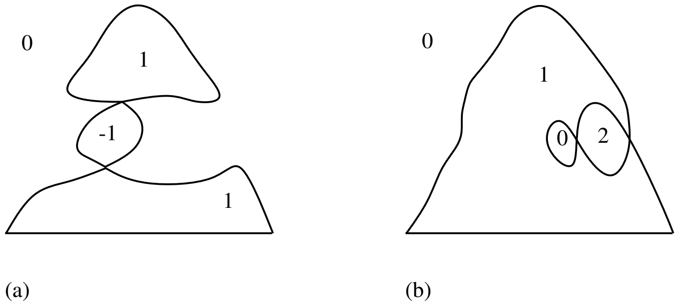


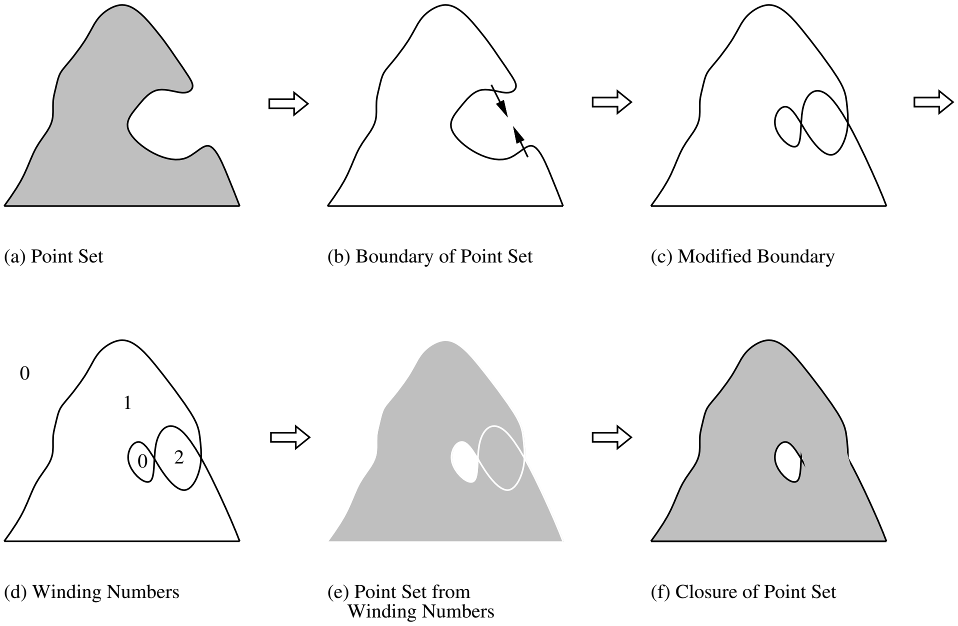
Figure 5.4: Classification of boundary elements using winding numbers.

5.2 Defining the Unary Shape Operations

In this section, we define the unary shape operations: unary union, unary intersection, and generalized unary intersection. Unary union and unary intersection are special cases of the *generalized unary intersection.* Therefore, we first define the generalized unary intersection, and then define the two special cases.

Intuitively, the nth generalized unary intersection of a surface is the set of points that are enclosed by the surface at least n times. The value of n can be an integer value since this corresponds to the range of the winding numbers. However, we are interested in bounded point sets, and thus restrict n to be greater than zero. We can give a more formal definition of the generalized unary intersection as follows.

Definition 5.3 *The* n**-th unary intersection** *of one or more oriented 2-manifold surfaces* $\mathcal{S}$,



(a) Point Set

(b) Boundary of Point Set

(c) Modified Boundary

(d) Winding Numbers

(e) Point Set from Winding Numbers

(f) Closure of Point Set

Figure 5.5: A point set view of the unary union operation.

which we denote as $\cap^n(\mathcal{S})$*, is the closure of the set of points with winding numbers greater or equal to* n*,*

$$k\{p \mid p \notin \mathcal{S}, w(\mathcal{S}, p) \geq n\}.$$

The unary union of a surface is the closure of the set of points that are enclosed at least once by that surface. We can then define the unary union in terms of the generalized unary intersection.

Definition 5.4 *The* **unary union** *of one or more oriented 2-manifold surfaces is the* 1*st unary intersection of those surfaces.*

Figure 5.5(a) shows the original solid from Figure 5.2(a). We can then modify the boundary (b) as we did before to produce the self-intersecting surface in Figure 5.5(c). We then compute the unary union of that surface by calculating the winding numbers (d), producing the set $\{p \mid p \notin \mathcal{S}, w(\mathcal{S}, p) \geq n\}$ (e), and finding the closure of the set (f).

We can also define the unary intersection. The unary intersection of a surface is the closure of the set of points that are enclosed at least twice by that surface. We can also define the unary intersection in terms of the generalized unary intersection.

Definition 5.5 *The* **unary intersection** *of one or more oriented 2-manifold surfaces is the* 2*nd unary intersection of those surfaces.*

The operations defined by the generalized unary intersection, including unary union and unary intersection, constitute the unary shape operations.

5.3 Properties of Unary Shape Operations

It is important that the unary shape operations produce the intended results for the intended domain. This section introduces properties of the unary shape operations and presents informal proofs of their correctness.

The unary union ensures that any self-intersections of a surface are resolved. This next theorem demonstrates that the unary union does not alter a valid solid, since its surface contains no self-intersections.

Theorem 5.1 *Any valid solid is identical to the unary union applied to its surface,*

$$\cap^1 (\partial S) = S.$$

Proof. The winding number of any point in the interior of a valid solid is 1, and in the exterior of the solid is 0. Therefore, the set

$$\{p \mid p \notin \mathcal{S}, w(\partial S, p) \geq\}$$

is identical to the interior of the solid. Since we are representing regular sets, the solid is equal to the closure of its interior, and thus, equal to the unary union.

$$S = k(i(S)) = \cap^1 (\partial S).\square$$

Since a valid solid cannot contain any self-intersections, its unary intersection is empty.

Theorem 5.2 *For $n > 1$, the nth unary intersection applied to the surface of a valid solid is empty,*

$$\textit{for}\ n > 1,\ \cap^n (\partial S) = \emptyset.$$

Proof. Again, the winding number of any point in the interior of a valid solid is 1, and in the exterior of the solid is 0. Therefore, the set

$$\{p \mid p \notin \mathcal{S}, w(\partial S, p) > 1\}$$

is empty, as is its closure.

$$\cap^1 (\partial S) = k(\emptyset) = \emptyset.\square$$

The unary intersection follows as a special case of the previous theorem. We can state this as a corollary.

Corollary 5.1 *The unary intersection applied to the surface of a valid solid is null,*

$$\cap^2(\partial S) = \emptyset.$$

Theorem 5.3 *The n-th unary intersection of one or more oriented 2-manifold surfaces $\mathcal{S}$ is a regular set.*

Proof. The set

$$\{p \mid p \notin \mathcal{S}, w(\mathcal{S}, p) \geq n\}$$

defines an open set of points from the interior of the surface. It cannot contain any isolated points, line segments, or "dangling" polygons. The closure of this set is therefore a regular set.□

If the oriented 2-manifold surface is semi-analytic, then the resulting regular set will also be semi-analytic. Therefore the regular set produced will also be an r-set.

Theorem 5.4 *Every set of oriented 2-manifold surfaces $\mathcal{S}$ has a unique n-th unary intersection.*

Proof. Since every point p not on $\mathcal{S}$ has a unique winding number $w(\mathcal{S}, p)$ with respect to the oriented surface, the set

$$\{p \mid p \notin \mathcal{S}, w(\mathcal{S}, p) \geq n\}$$

is unique. The n-th unary intersection is the closure of this set, and since the closure of any point set is unique, the n-th unary intersection must be also be unique.□

Theorem 5.5 *R-sets are closed under local operations followed by unary shape operations.*

Proof. Given an r-set, we can construct a boundary description of the surface of the r-set. (Indeed, we generally represent the r-set by the bounding surface, or by primitive solids for which we have boundary descriptions.) If the r-set has a manifold boundary, the winding number will be defined for any point not on the boundary. If the r-set has a nonmanifold boundary, then we consider it to be an immersion of several 2-manifold boundaries. Following the definition of r-sets, the boundary will be semi-analytic.

Local operations transform the boundary description, while maintaining its boundedness, continuity, orientability, and semi-analyticity. From the definition of generalized unary intersection, the set

$$\{p \mid p \notin \mathcal{S}, w(\mathcal{S}, p) \geq n\}$$

(where $\mathcal{S}$ is the transformed boundary and $n > 0$) produces the interior points of the volume enclosed n or more times by the boundary description, excluding any of the points on the bounding

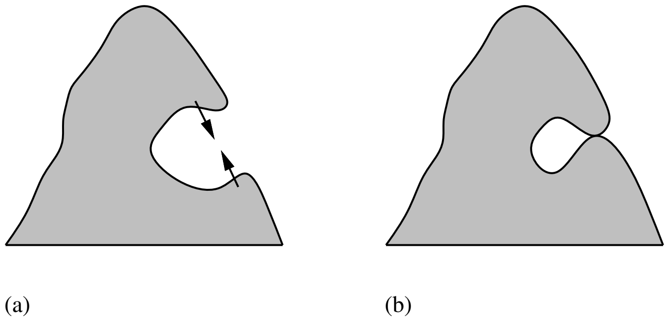


Figure 5.6: Nonmanifold condition at a vertex due to local operations.

surface(s). We then take the closure of this set. Since the resulting set is bounded, closed, regular, and semi-analytic it is an r-set.□

We can modify the geometry of a solid such that a nonmanifold condition is created (Figure 5.6). The unary shape operations will not modify the geometry of this solid, but will compute and record the additional topological adjacencies.

Theorem 5.6 *2-manifold r-sets are not closed under local operations followed by unary shape operations.*

Proof. This may be proven by example. Given the 2-manifold r-set in figure Figure 5.6(a), we can modify the boundary description as shown. The resulting r-set does not have a 2-manifold surface. □

5.4 The Inversion Operation

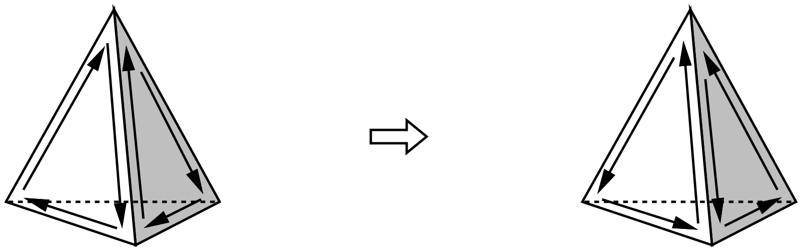


Figure 5.7: A tetrahedron and its inversion.

The *inversion* of the bounding surface of a solid describes a finite void or absence of material. When an inverted solid is combined with the original solid, the two objects “neutralize” each other. When considering valid solids, an inverted solid acts as a *negative object*, and the inversion operation is identical to the negation operator, as defined by Braid [3]. The inversion operation and inverted solids are useful for computing the Boolean difference of solids, to be discussed in the next section.

Definition 5.6 *The* **inversion** *of one or more oriented surfaces $\mathcal{S}$, denoted $-(\mathcal{S})$, is the same surfaces with their orientations reversed.*

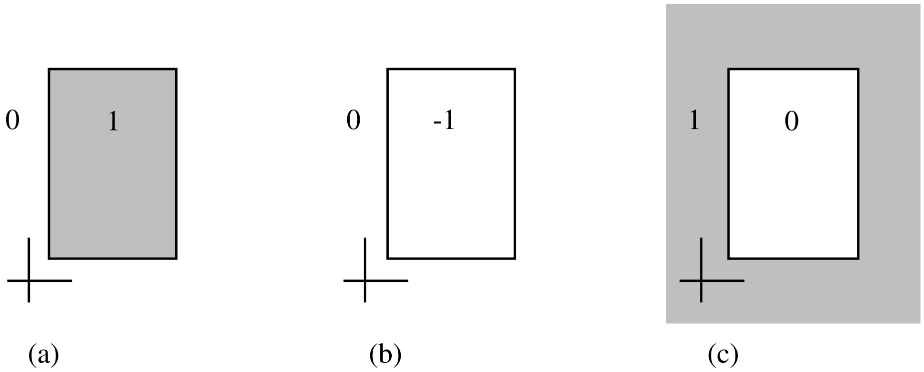


Figure 5.8: A 2-D solid, its inversion, and its complement.

The inversion operation switches the orientation of the bounding surfaces of a solid. The inverted surfaces will bound the same points as the original surfaces. Since the orientation of the surfaces of the solid has changed, the sign of the winding numbers of all points relative to the surfaces will also change. It is important to note that the inversion of a solid is *not* the same as the complement of that solid. To illustrate this difference, the inversion and complement of a two dimensional solid is shown in Figure 5.8.

5.5 Boolean Operations

Nearly all available solid modelers provide regularized Boolean operations (union, intersection, and difference) for combining solids.

The union of two solids is the boundary of the volume contained by either of the objects. The intersection is the boundary of the volume contained in both. The difference of two solids is the boundary contained in the first solid, but outside of the second.

Boolean operations can be included in the modeling scheme in at least two ways. The two methods produce identical results for valid representations, but may produce different results for representations that are topologically valid, but not geometrically valid.

The first method is straightforward. Boolean operations require that the two input solids are valid (both topologically and geometrically). Since the local operations may produce a solid description that is not geometrically valid, we can use the unary shape operations to produce input solids that are valid. The Boolean operations can then be applied in the usual way.

The second method uses the unary shape operations to compute the Boolean operations directly. This section presents the methods used to compute the Boolean operations, and contains proof of the correctness of this method.

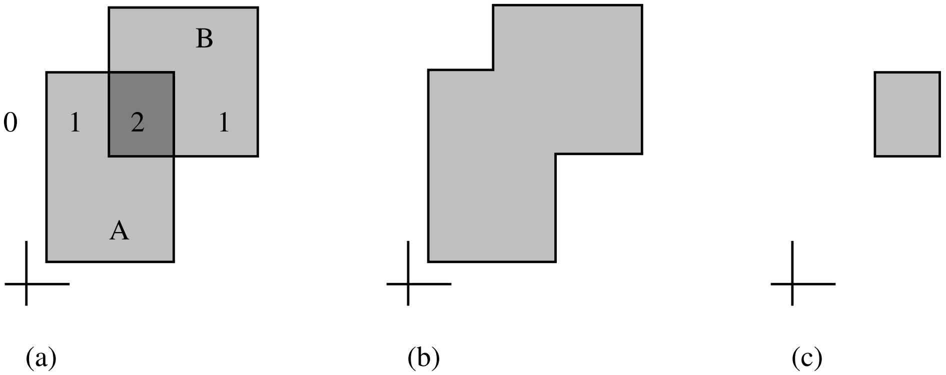


Figure 5.9: Solids A and B, their union $A \cup^* B$, and their intersection $A \cap^* B$.

Boolean union is computed by merging the two (valid) solids and computing the unary union for the surfaces of the merged solid. This requires multi-shell solids. The `merge_solids` operation is described in Chapter 4, and is denoted by $+$.

Theorem 5.7 *The regularized union of two solids is equal to the unary union of the combined surfaces of A and B, $A \cup^* B = \cap^1(\partial A + \partial B)$.*

Proof. The regularized Boolean operations are only defined for valid solids. Therefore, the proofs presented here only discuss those cases. The regularized union produces the closure of the set of points that are in either the interior of A or the interior of B. Since the regularized union is only defined for valid solids, then we know that $A = \cap^1(\partial A)$ and $B = \cap^1(\partial B)$. The winding number of points in the interior of A and the exterior of B will be 1. The winding number of points in the exterior of A and the interior of B will also be 1. The winding number of points in the interior of A and the interior of B will be 2. The winding number of points in the exterior of both A and B will be 0. From the definition of unary union, the set

$$\{p \mid p \notin (\partial A + \partial B), w((\partial A + \partial B), p) \geq 1\}$$

will contain the points that are in the interior of A, the interior of B, or both, and do not lie on the boundary of either A or B. The unary union produces the closure of this set of points, adding all the points that are on the boundary of A and the points that are on the boundary of B. Thus $A \cup^* B$ and $\cap^1(\partial A + \partial B)$ are equal.□

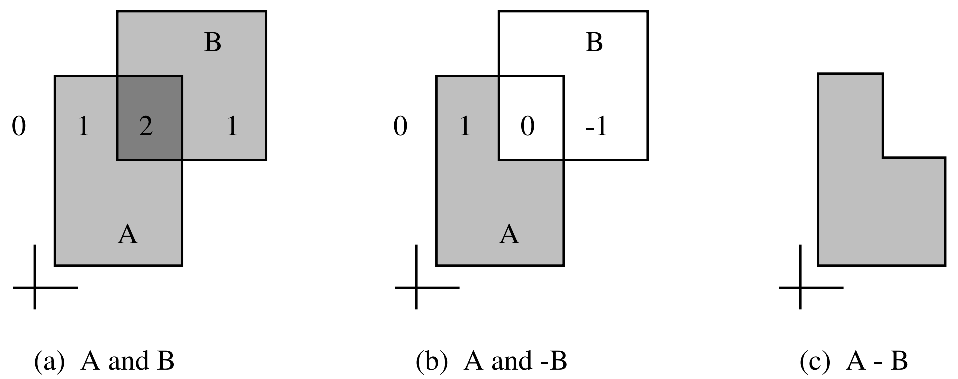
Boolean intersection can be computed by merging the two solids and computing the unary intersection of the merged solid.

Theorem 5.8 *The regularized intersection of two solids is equal to the unary intersection of the combined surfaces of A and B, $A \cap^* B = \cap^2(\partial A + \partial B)$.*

Proof. This proof is similar to the previous one. The regularized intersection produces the closure of the set of points that are in both the interior of A and the interior of B. The winding number of points in the interior of A and the interior of B will be 2. The winding number of all other points, if defined, will be 1 or less. From the definition of unary union, the set

$$\{p \mid p \notin (\partial A + \partial B), w((\partial A + \partial B), p) \geq 2\}$$

will contain the points that are both in the interior of A. The unary intersection produces the closure of this set of points. Thus $A \cap^* B$ and $\cap^2(\partial A + \partial B)$ are equal.□



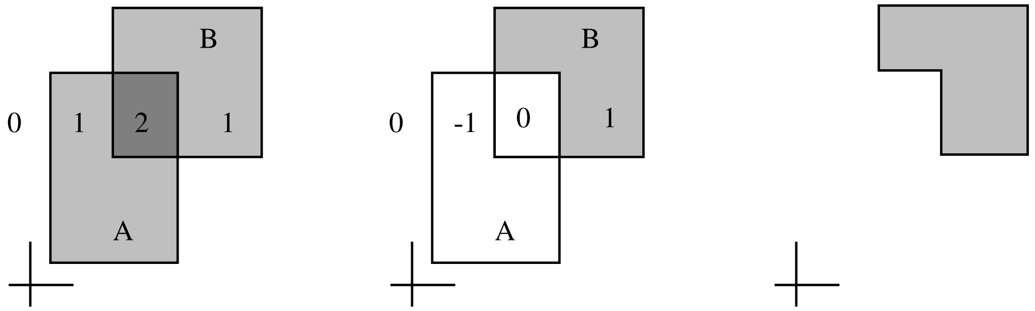
(a) A and B (b) A and -B (c) A - B

Figure 5.10: Solids A and B, A and the inverse of B, and the difference $A -^* B$.

Boolean difference can be computed by inverting the solid to be subtracted, merging the solids, and computing the unary union.

Theorem 5.9 *The regularized difference of two solids, A and B, is equal to the unary union of the combined surface of A and the inverted surface of B, $A -^* B = \cap^1(\partial A + (-\partial B))$.*

Proof. The regularized difference produces the closure of the set of points that are in the interior of A but not in the interior of B. Since B is given to be a valid solid, $B = \cap^1(\partial B)$, and the winding number of each of the points in the interior of B is 1. When B is inverted, the winding numbers switch sign. Once the surfaces of the solids are merged, the winding numbers of points in the interior of A and the exterior of B will be 1. The winding number of points in the exterior of A and the interior of B will be -1. The winding number of points in the interior of A and the



(a) A and B (b) -A and B (c) B - A

Figure 5.11: Solids A and B, A and the inverse of B, and the difference $A-^* B$.

interior of B will be 0. Finally, the winding number of points in the exterior of both A and B will be 0. From the definition of unary union, the set

$$\{p \mid p \notin (\partial A + \partial B), w((\partial A + (-\partial B)), p) \geq 1\}$$

will contain the points that are in both the interior of A and the exterior of B. The unary union produces the closure of this set of points. Thus $A-^* B$ and $\cap^1 (\partial A + (-\partial B))$ are equal.□

The (traditional) Boolean operations are not defined for (surfaces of) solids that are topologically valid, but not geometrically valid. However, Boolean operations computed with the unary shape operations produce identical resulting solids from valid input solids as the (traditional) Boolean operations, and produce valid solids from input solids that are topologically valid, but geometrically invalid.

Using the unary shape operations to compute Boolean operations increases the number of the comparisons (you must compare all elements with each other). The results are guaranteed to be valid with topologically valid input.

5.6 Computing the Unary Shape Operations

We can produce the unary union, unary intersection, regularized Boolean union, intersection, and difference, all using the generalized unary intersection and the inversion operation. This section is devoted to a sketch of an algorithm for the computation of the generalized unary intersection for polyhedral surfaces.

The algorithm for computing the generalized unary intersection draws on existing methods for computing Boolean operations, specifically those presented by Requicha and Voelcker [45], Mäntylä [35], Laidlaw, et al [29], and Karasick [25].

Maintaining Consistent Face Descriptions

Local operations can create inconsistent face descriptions. The types of inconsistency and ill-formedness find several forms. For example, some of the vertices of the face may not all lie in the same plane, or may lie outside of the boundary of the face. The vertices of the face may be arranged such that the edges joining them self-intersect. We may also find the an internal loop of a face in an inverted orientation, such that it does not define a hole in that face. We can then see that the topological description of a face may be inconsistent with the geometry assigned to the elements of that face.

Consistent face descriptions are required as preconditions for proper operation of the unary shape operations. Since we allow the user the ability to construct inconsistent face descriptions, we must have a method for removing these inconsistencies. We have two basic approaches.

The first approach is to detect an inconsistency and alert the user. Inconsistencies are generally the result of an error due to the action of a user or a user's routine. The user should be aware of the problem, and generally knows best how to solve it.

The second approach is to automatically remove the inconsistency. This is useful when the differences in the possible results are not meaningful. We introduce operations to convert inconsistent face descriptions to consistent ones. This may be accomplished (in the worst case) by triangulating the face.

An Algorithm for the Generalized Unary Intersection

The definitions of the unary shape operations presented map from a surface to a point set. Since we are interested in representing point sets by their bounding surfaces, we would like the unary shape operations to map from surface descriptions to surface descriptions (see Figure 5.12). We can then say that the n-th unary intersection of a surface is the surface that separates points with winding numbers less than n from points with winding numbers n or greater.

Once face descriptions are made consistent, the generalized unary intersection can be computed using the following algorithm:

- Find all intersections between the faces, edges and vertices. (See Figures 5.13 to 5.16.)
- Modify the boundary representation with Euler operators to represent these intersections explicitly in the topology, and reorganize the face adjacencies about the intersection. This reorganization considers the normals of the faces, and modifies the adjacencies of the faces about the edge of intersection such that the one pair of adjacent faces is (locally) contained within the surface of the other pair of adjacent faces. The reorganization of adjacencies is illustrated in Figure 5.17.
- Calculate the winding numbers of points within each separated shell (See Figure 5.18). This

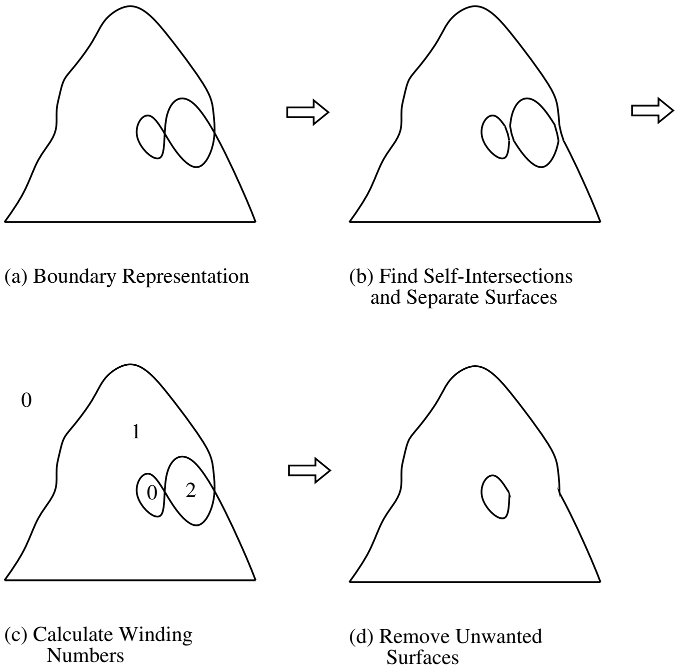

(a) Boundary Representation
(b) Find Self-Intersections and Separate Surfaces
0
1
0
2
(c) Calculate Winding Numbers
(d) Remove Unwanted Surfaces

Figure 5.12: Computing the unary union.

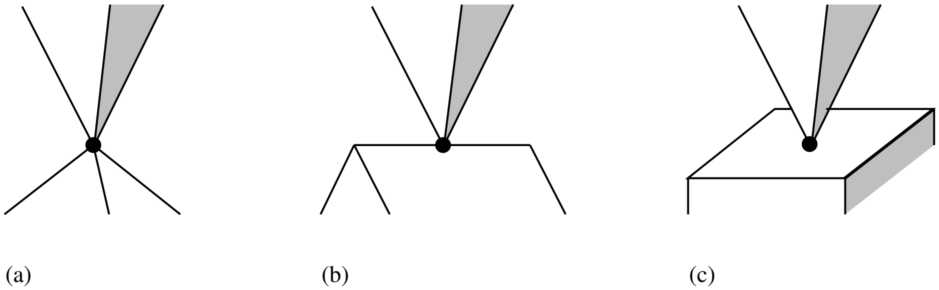


Figure 5.13: A vertex intersecting a vertex, an edge, and a face.

calculation usually requires ray casting. Some of the ray casting tests can be avoided by making use of local properties of connectivity and the sense of normals and edges. For example, when two faces intersect and the adjacencies are reorganized, then the relative winding numbers are known. The calculation of the winding number of points within one of the shells will determine the winding number of the points within the other shell.

- Remove all unnecessary shells. The n-th generalized unary intersection requires the removal of all shells that do not separate points with winding numbers less than n from those with winding numbers n or greater.

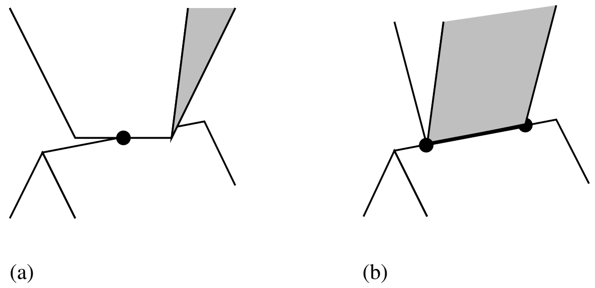


Figure 5.14: An edge intersecting an edge.

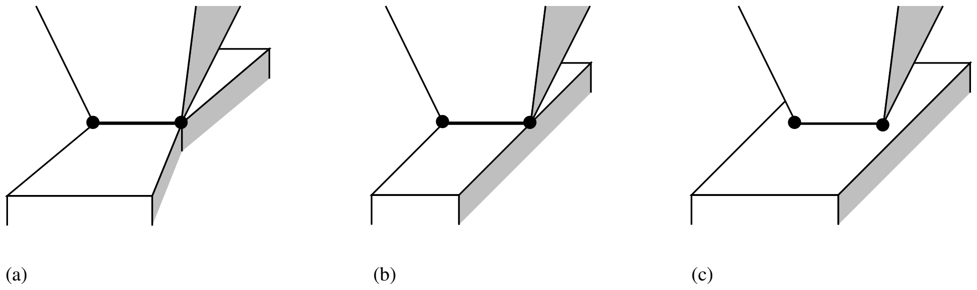


Figure 5.15: An edge intersecting a face.

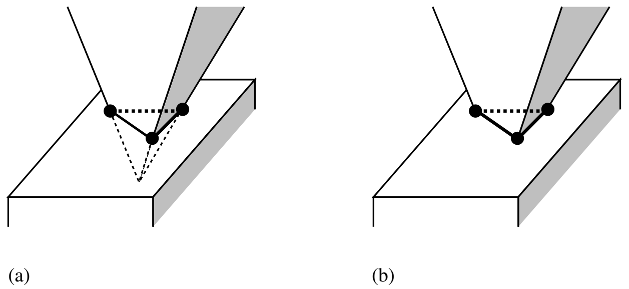


Figure 5.16: A face intersecting a face.

The algorithm for generalized unary intersection allows us to compute the unary union and unary intersection by simply computing the first unary intersection and second unary intersection. It also allows us to compute the regularized Boolean union, intersection. Computing the regularized Boolean difference may be accomplished by implementing an inversion operation, and using the unary union. An efficient implementation of the inversion operation follows directly from the definition.

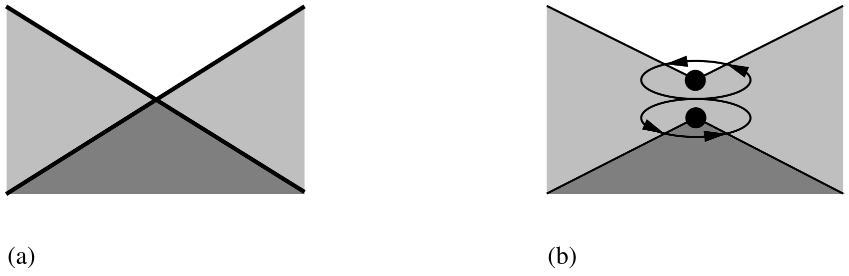


Figure 5.17: Reorganization of the adjacency relations for two intersecting faces.

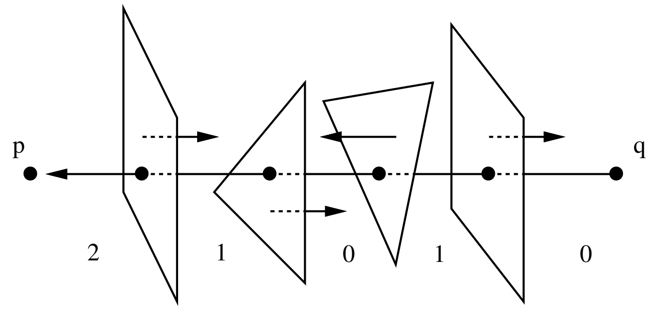


Figure 5.18: Calculating the winding number of a point p.

Chapter 6

Modeling With Topologically Valid Solids

In this chapter, we argue for a new approach to solid modeling using boundary representations. In order to present our approach, we characterize the modeling space, representation space, and the mappings of the modeling operations. We then characterize the existing approaches to solid modeling and contrast them with the new approach.

6.1 Representation Schemes

The notion of representation scheme and the characterization of the constructive solid geometry scheme were introduced by Requicha [43].

Definition 6.1 *A* **representation scheme** *is a relation $S : M \rightarrow R$. The domain of S is denoted by D and the image of D under S is V.*

Figure 6.1 illustrates the idea of a representation scheme. Models are abstract (existential, representation-independent) mathematical entities that capture the physical essence of rigid solid objects. Representations (models) are symbol structures used to represent solids. In Figure 6.1, R is the set of possible symbol structures, T is the set of symbol structures that are topologically valid, and V is the set of symbol structures that are both topologically and geometrically valid.

For the purposes of this discussion, we are interested in operations that allow us to create and manipulate these representations. In terms of the illustration, these operations map from subsets of R into other subsets of R.

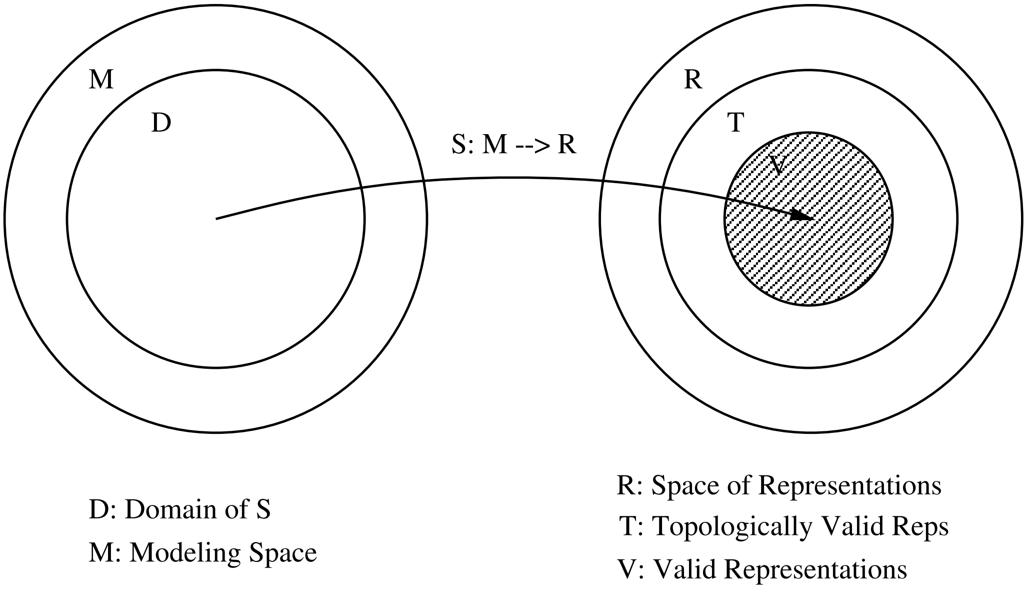


Figure 6.1: A representation scheme.

6.2 Traditional Approaches

Constructive solid geometry uses a very direct representation scheme. The primitive solids are a small set of valid representations of solids. The regularized Boolean operations combine valid models to form new valid models, i.e.

$$\text{Boolean Operations} : V \to V.$$

The approaches to boundary representation come in two forms. The first is a variation on the constructive solid geometry approach explained above. This uses a boundary representation of solids, valid primitive solids, and an evaluated form of the regularized Boolean operations to combine solids, i.e.

$$\text{Boolean Operations} : V \to V.$$

A second boundary representation approach has also been used. This allows the use of local operations on geometry and Euler operations on the topological representation, i.e.

$$\text{Euler Operations} : T \to T,$$

$$\text{Local Operations} : T \to T,$$

Since these operations may cause models to self-intersect, the user is responsible for ensuring that the validity of the model is maintained. The regularized Boolean operations are generally provided, but require valid models as input, i.e.

$$\text{Boolean Operations} : V \rightarrow V.$$

Even though local operations permit invalid results, these operations have shown their usefulness and are provided in many commercial solid modeling systems. Boolean operations also have a long history of use in the solid modeling literature. Because of their distinct uses, it is desirable to have both local operations and Boolean operations in a solid modeling system.

6.3 A New Modeling Approach

We propose a new modeling approach. Local operations on geometry and Euler operations produce topologically valid models, i.e.

$$\text{Euler Operations} : T \rightarrow T,$$

$$\text{Local Operations} : T \rightarrow T.$$

These models may have self-intersecting surfaces. The unary shape operations are then used to convert any topologically valid model into a (topologically and geometrically) valid model, i.e.

$$\text{Unary Operations} : T \rightarrow V.$$

The Boolean operations are implemented in terms of unary shape operations. Therefore, they also map topologically valid representations into valid representations,

$$\text{Boolean Operations} : T \rightarrow V.$$

This approach provides the utility of using both local and Boolean operations, while assuring the topological and geometric validity of the generated models.

Part II

Generative Geometric Design

The second part of the thesis explores generative geometric design within a grammatical paradigm.

The shape grammar formalism of Stiny [52] has been particularly influential in architecture. This formalism uses a representation of "shapes". Rewrite rules are defined by two shapes, the first, when matched, is removed and replaced by the second. Boundary solid grammars draw on the ideas of shape grammars, while using a solid modeling representation and parametric rule matching and application.

Much of the application of shape grammars has been for composition and analysis in architectural design, including Palladian villas [53], Frank Lloyd Wright's Prairie Houses [28], Queen Anne houses [17, 18], bungalows [12], modern Italian apartments (after the Casa Giuliani Frigerio) [16], and Japanese tea houses [27]. Additional shape grammars have been written to generate Chinese ice ray designs [51], Moghul gardens [54], and chair-back designs [26].

Graph grammars and L-systems, although not geometric representations, have been used to generate graphs or arrays that are mapped to polygons, lines, or primitive solids. Much of this work has been directed at describing biological growth, especially of plants and trees [40, 39, 10].

Checking match conditions for applying a solid grammar rule is equivalent to recognizing features of a solid. Some of the recent feature recognition research uses methods similar to those used in boundary solid grammars. De Floriani used a face-based boundary representation and features described as topology graphs [9]. Pinilla used an edge-based representation augmented with arcs representing primitive geometric relations (parallel and perpendicular), and located subgraphs generated from a graph grammar description of features [38]. The method of feature recognition presented here allows a flexible description of features.

Chapter 7

Boundary Solid Grammars

The term "grammar" is used here as the generative specification of families of models. A grammar may be said to consist of a representation, an initial model in that representation, and a set of rules. A rule is applicable to a model if it can match on features of that model, in which case it transforms the model to a new form. Rules may be applied to an initial model, and to models produced thereof, to produce new models. Thus, applying different rules, and applying them in different ways, allows us to generate a family or *language* of different models.

The representation used in boundary solid grammars is a boundary representation of solids. The initial model is then a (possibly empty) collection of solid models. The rules of a boundary solid grammar modify the solids to generate languages of solid models. This chapter introduces the boundary solid grammar formalism, detailing the representation, initial solids, and the formulation of solid rules. It then describes boundary solid grammars and the languages that they generate. A simple grammar is presented to illustrate the ideas explained in this chapter. Additional grammars are presented in Chapter 8.

7.1 Representation of Solids

A boundary solid grammar uses a boundary representation of solid objects. The topology is represented as a graph composed of nodes and arcs. The nodes of these graphs are topological elements, and the arcs represent the adjacencies between elements. The geometry used in this thesis consists of vertex coordinates for polyhedral solids. The topology and geometry together define a boundary representation solid model.

The boundary solid grammar detailed here uses a representation consisting of:

- a *topology* graph with *vertex, edge-half, loop, face, shell* and *solid nodes*, corresponding to the generalized split-edge data structure introduced in Chapter 2, and *arcs* representing their adjacencies;

- coordinate *geometry* in $\Re^3$ associated with each vertex; and
- sets of *labels* associated with topology nodes, as desired; and
- a *state* associated with each instance of the representation.

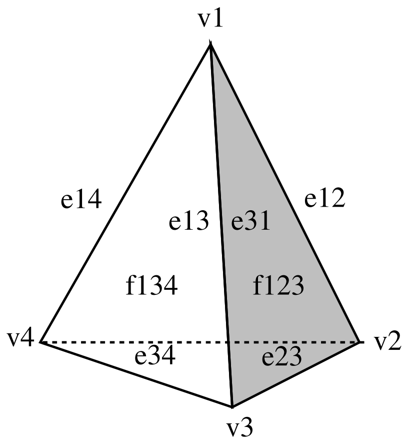


Figure 7.1: A simple solid.

In order to explain our notation for our boundary representation, we will present the representation in four steps. The first will define the graphs of topological elements and their adjacencies. The second will define the coordinate geometry associated with vertices, providing a complete definition of the boundary representation. The third step will define how to associate non-geometric data to topological elements. The final step will define how to associate a current state to the representation.

To illustrate this representation on a concrete example, we will show the four incremental representations of the tetrahedron shown in Figure 7.1. In the figures, the vertices are labeled v1 to v4, the edge-halves are labeled by the vertices at either end (the "from" vertex listed first), the loops and faces are labeled by the cycle of vertices about it (in clockwise order as viewed from outside), the shell is labeled sh1, and finally the solid is labeled s1. The visible vertices, edge-halves, and faces are labeled in this fashion in Figure 7.1.

Definition 7.1 *A* **topology graph** *over the alphabet* $\Sigma_{node} \cup \Sigma_{arc}$ *is a tuple* $\mathcal{TG} = \langle K, (\rho_a)_{a \in \Sigma_{arc}} \rangle$, *where*

1. $\Sigma_{node} = \{\textsc{vertex}, \textsc{edgehalf}, \textsc{loop}, \textsc{face}, \textsc{shell}, \textsc{solid}\}$ *is the alphabet of topological element types;*

2. The following specifies the alphabet of relations between topological elements:

$$\Sigma_{arc} = \left\{ \begin{array}{llllll} vertex\_l, & edgeh\_l, & loop\_f, & face\_sh, & shell\_solid, & solid\_shell, \\ vertex\_eh, & cw\_eh, & next\_face\_l, & next\_shell\_f, & outside\_sh, & \\ next\_vertex\_use, & ccw\_eh, & loop\_eh, & prev\_shell\_f, & connected\_sh, & \\ & other\_eh, & loop\_v, & face\_l, & inside\_sh, & \\ & edgeh\_v, & & & shell\_f & \end{array} \right\}$$

3. K_t with $t \in \Sigma_{node}$ specifies a finite set of nodes of element type t. The sets K_t are disjoint. $K = \bigcup_{t \in \Sigma_{node}} K_t$ is the set of all nodes; and

4. $\rho_a \subseteq K \times K$ are relations (arcs) over K, one for each a in Σ_{arc}.

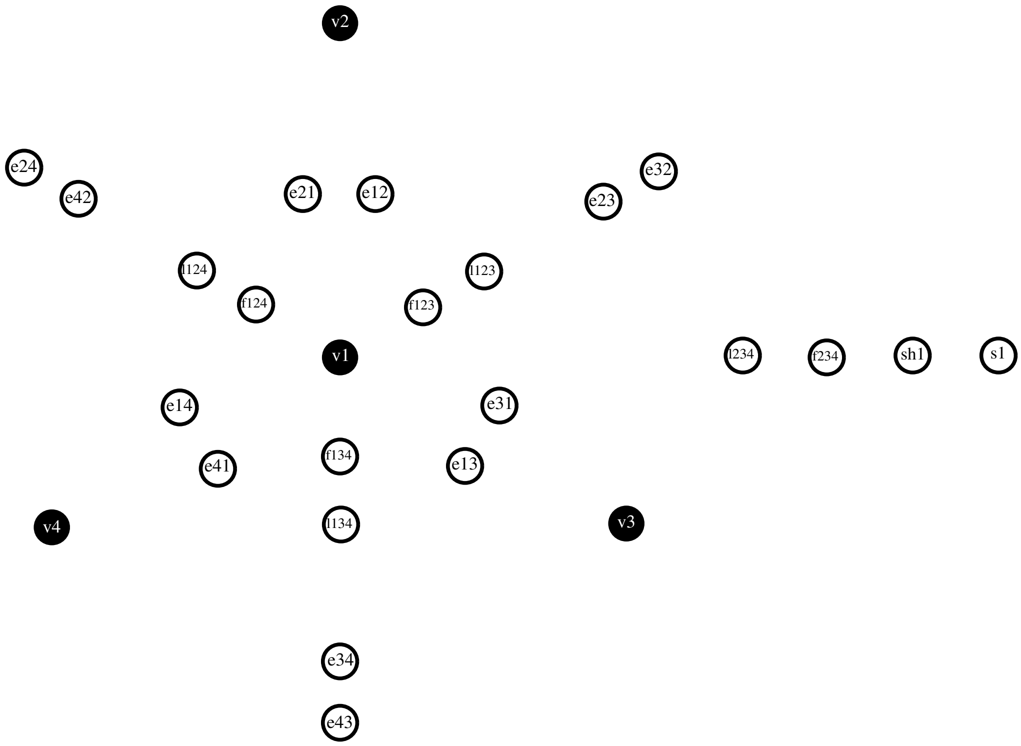


Figure 7.2: The nodes of a topology graph for a tetrahedron.

The topology graph defines a graph of topological elements and their adjacencies. The nodes are the topological elements of our generalized split-edge representation. The arcs are the topology

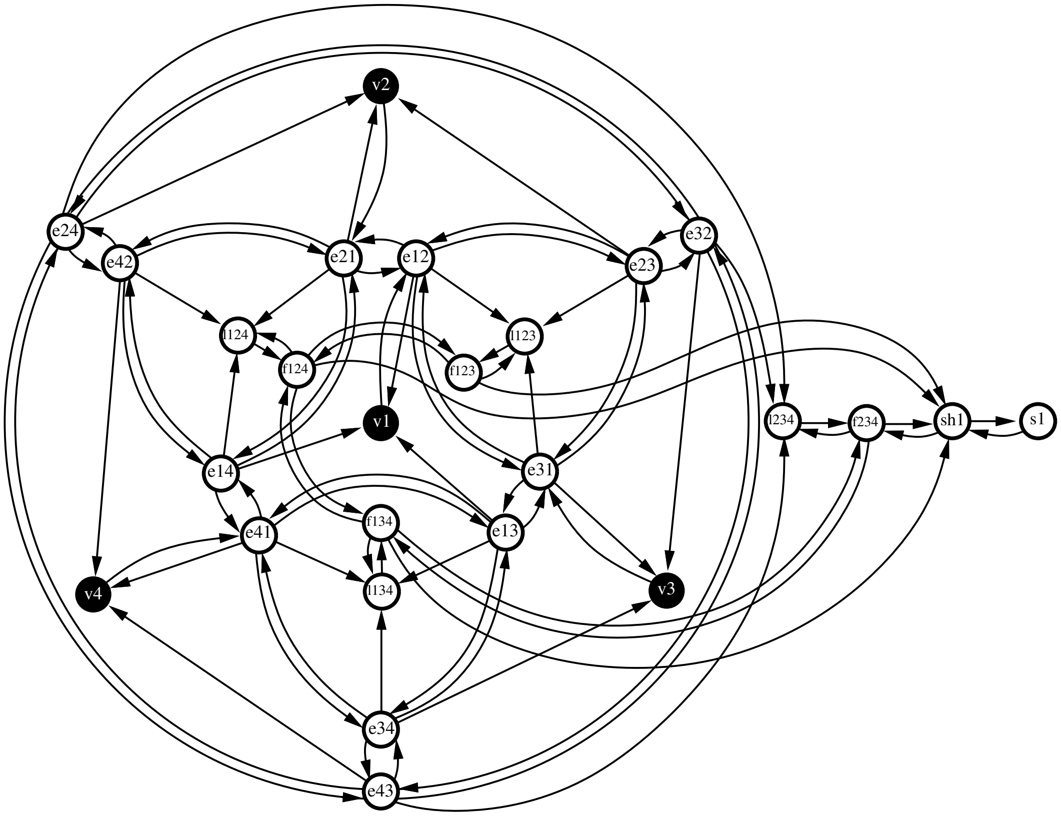


Figure 7.3: The topology of a tetrahedron represented with a topology graph.

graph are the topological adjacencies between these elements. For the details of these topological elements and their adjacency relations, refer to Chapter 3.

The nodes of the topology graph of the tetrahedron in Figure 7.1 are shown in Figure 7.2. They include the four vertices (the black nodes), twelve edge-halves (two for each of the six edges), four loops and four faces (each face has a single loop of edges and vertices), one shell (surface), and one solid.

The topological adjacencies are added as directed arcs in Figure 7.3. Although the arcs are not labeled, it is not difficult to infer the adjacency relation from the node types. For example, the arc from the shell $sh1$ to the solid $s1$ containing it is a *shell_solid* arc, and the arc from $s1$ to the first shell $sh1$ is a *solid_shell* arc.

Definition 7.2 *A* **boundary graph** *over the alphabet* $\Sigma_{node} \cup \Sigma_{arc} \cup \Re^3$ *is a tuple* $\mathcal{BG} = \langle K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma \rangle$, *where*

1. $\langle K, (\rho_a)_{a \in \Sigma_{arc}} \rangle$ *is a topology graph; and*

2. $\gamma : K_{\text{VERTEX}} \rightarrow \Re^3$ *is a function that maps vertex nodes to vertex coordinates.*

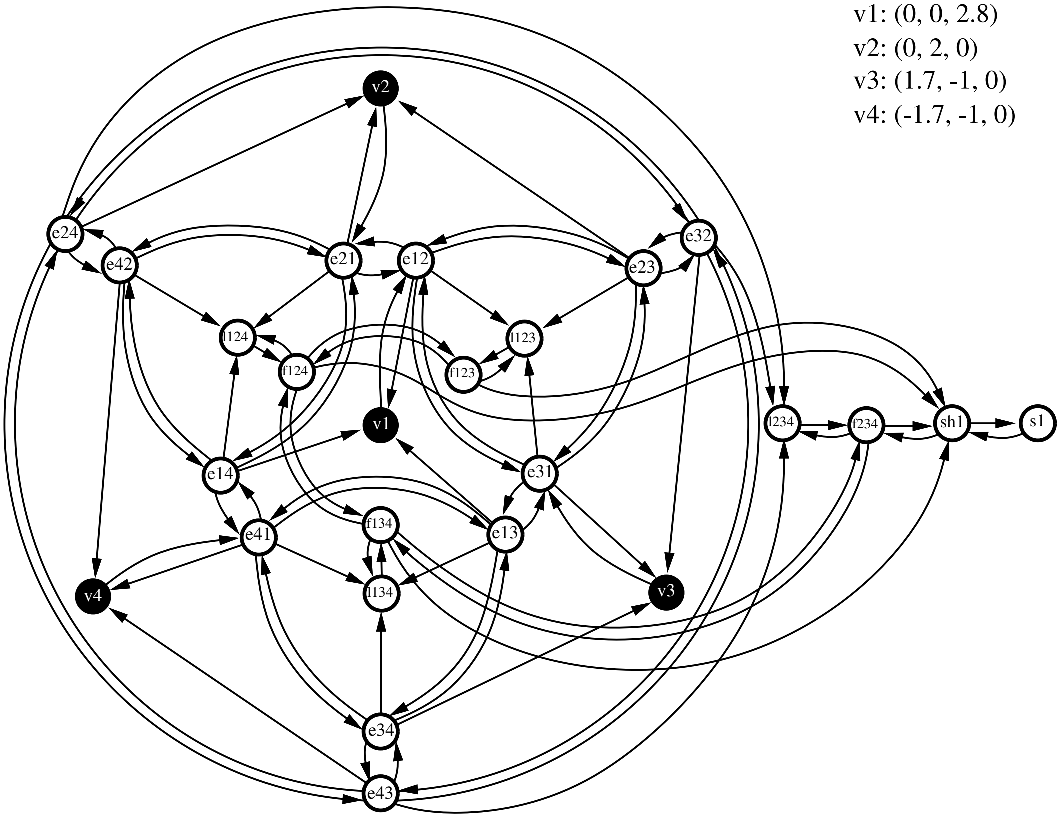


Figure 7.4: A tetrahedron represented with a boundary graph.

A boundary graph is a topology graph with coordinate geometry associated with each vertex. The boundary graph for our tetrahedron is shown in Figure 7.4. The mapping from vertices to vertex coordinates is shown in the upper right corner of the figure.

Definition 7.3 *A* **labeled boundary graph** *over the alphabet* $\Sigma_{node} \cup \Sigma_{arc} \cup \Re^3 \cup \Sigma_{label}$ *is a tuple* $\mathcal{LG} = \langle K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma, \lambda \rangle$, *where*

1. $\langle K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma \rangle$ *is a boundary graph;*

2. $\Sigma_{attribute}$ *is an alphabet of label attributes;* Σ_{value} *is an alphabet of label values;* $\Sigma_{label} = \Sigma_{attribute} \times (\Sigma_{value} \cup \Re)$ *is the set of labels;*

3. $\lambda : K \rightarrow 2^{\Sigma_{label}}$ *is a function from nodes to sets of labels;*

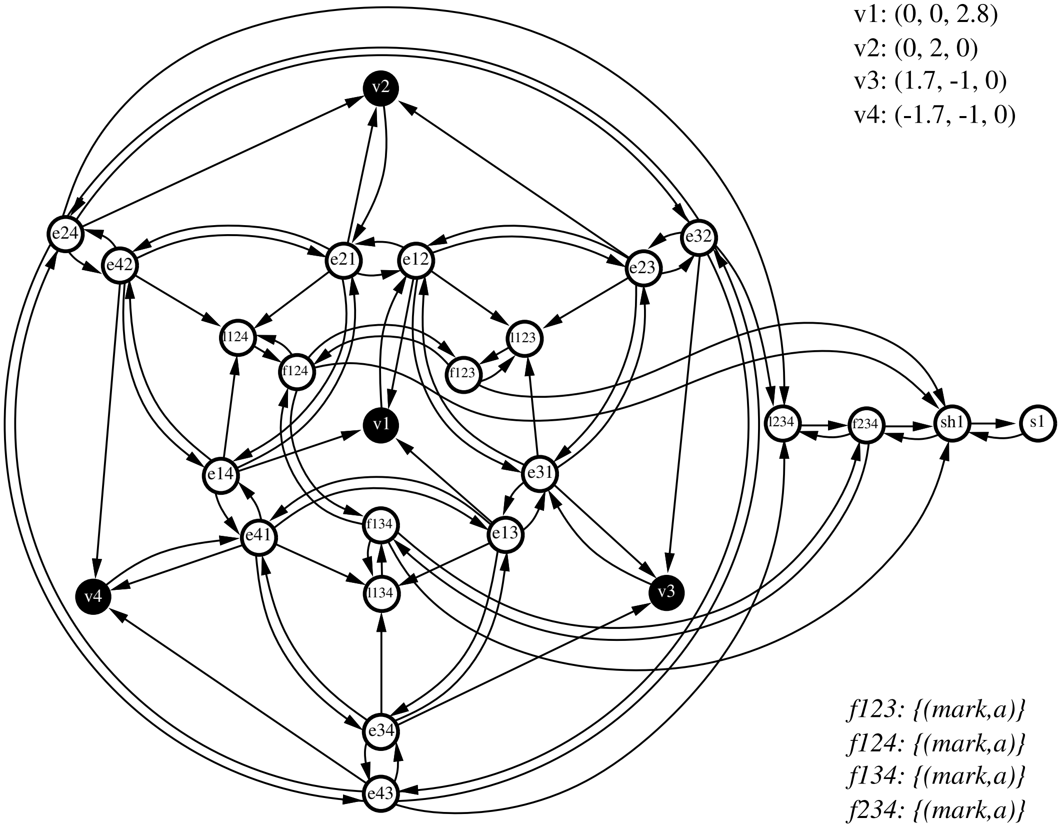


Figure 7.5: A tetrahedron represented with a labeled topology graph.

Labels provide a mechanism for associating non-geometric data with topological elements of solids. They have several uses in boundary solid grammars. Labels may be required as conditions in rules to restrict rule application. This is illustrated in an example grammar presented later in this thesis. Labels may be used to reduce search by marking conditions needed in future operations. They also allow completed constructions of solids to terminate, while prohibiting termination of incomplete or invalid solids.

In Figure 7.5, a “(mark,a)” label is associated with each of the four faces. This is an attribute-value pair to indicate that there is a “mark” of a type “a” on a face. These attribute-value pairs are also used, for example, to indicate the type of material from which a solid is composed. If we want to indicate that the tetrahedron in Figure 7.5 is made of steel, then we would use a label on the solid such that $\lambda(\text{s1}) = \{(\text{material, steel})\}$. This information could then be accessed for the computation of the mass of the solid. It could also be used in order to render the solid with the appropriate texture and color.

Finally, we associate a state, from a finite set of states, to the representation. We could refer to this final representation as a "finite state labeled boundary graph". However, we have chosen to use the shorter term, *b-graph*, to refer to this representation.

Definition 7.4 *A* **b-graph** *over the alphabet* $\Sigma_\star = \Sigma_{node} \cup \Sigma_{arc} \cup \Re^3 \cup \Sigma_{label} \cup S$ *is a tuple* $\mathcal{B} = \langle K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma, \lambda, \sigma \rangle$*, where*

1. $\langle K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma, \lambda \rangle$ *is a labeled boundary graph;*

2. $S = S_{intermediate} \cup \{start, done\}$ *is a finite set of states; and*

3. $\sigma \in S$ *is the current state.*

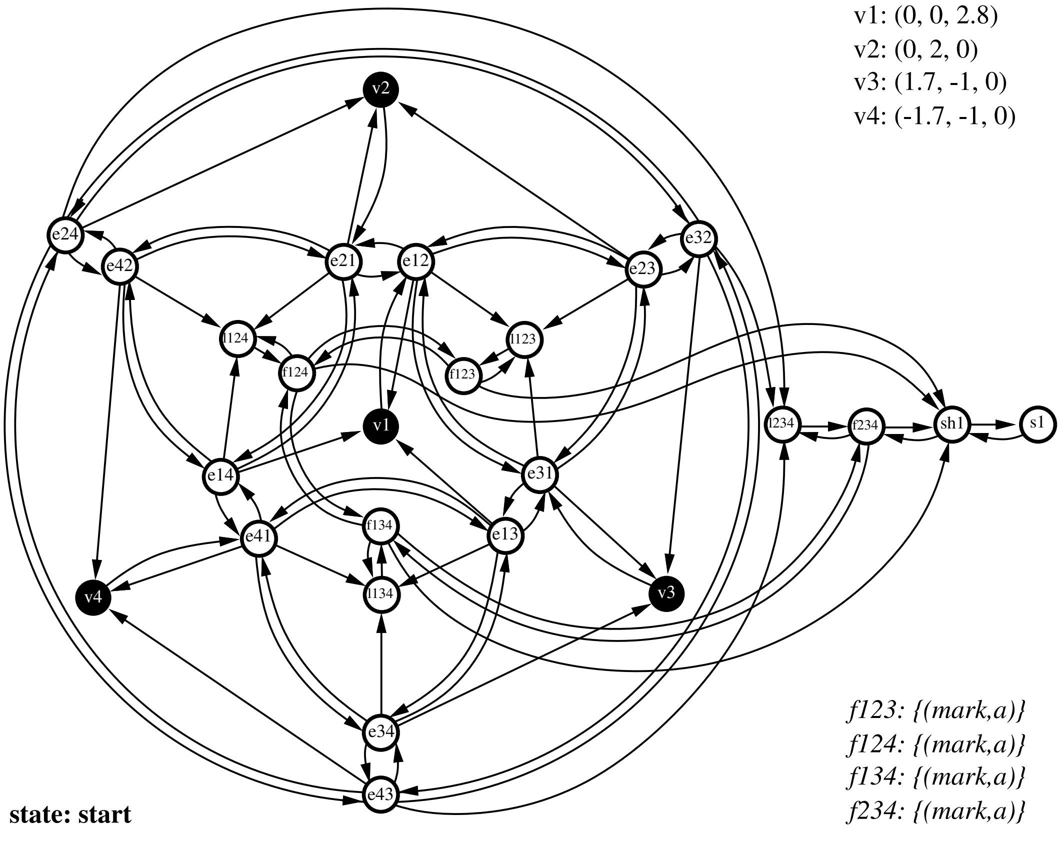


Figure 7.6: A tetrahedron represented with a b-graph.

A state is associated with each labeled boundary graph. The state is used to determine if a given b-graph is a member of the language of the grammar. This will be discussed in greater detail in the discussion of the language of a grammar, later in this chapter. This finite state mechanism is similar to the state mechanism found in the programmed grammars of Rosenkrantz [46, 47], and the programmed graph grammars of Bunke [5, 6].

The state is also useful as a label on the graph indicating or restricting which rules may apply at the current time. It is also a useful and computationally efficient mechanism for determining if the b-graph is a member of the language of a grammar. Figure 7.6 illustrates the representation of a b-graph of a tetrahedron.

Let $\mathcal{B}(\Sigma_\star)$ denote the set of all b-graphs over the alphabet $\Sigma_\star$, and $\mathcal{B}_\varepsilon$ denote the empty b-graph. The empty b-graph has a state $\sigma \in S$ as its current state, but contains no nodes, arcs, or labels.

Not all b-graphs represent rigid solid objects. A b-graph may have nodes and arcs that do not correspond to a physical solid. For example, we can construct a b-graph with an EDGEHALF node connected to more than one VERTEX node with *edgeh_v* arcs. This is clearly invalid, since an edge connects exactly two vertices. Even if such simple discrepancies are avoided, it is still possible to generate b-graphs that do not correspond to valid plane models and have faces with inconsistent orientations. Euler operators are used in order to construct only topologically valid b-graphs.

7.2 The Initial Solid

An initial solid of the boundary solid grammar formalism is a valid b-graph. This b-graph may be the empty b-graph, $\mathcal{B}_\varepsilon$, or it may be a b-graph consisting of an number of (arbitrarily complex) solids.

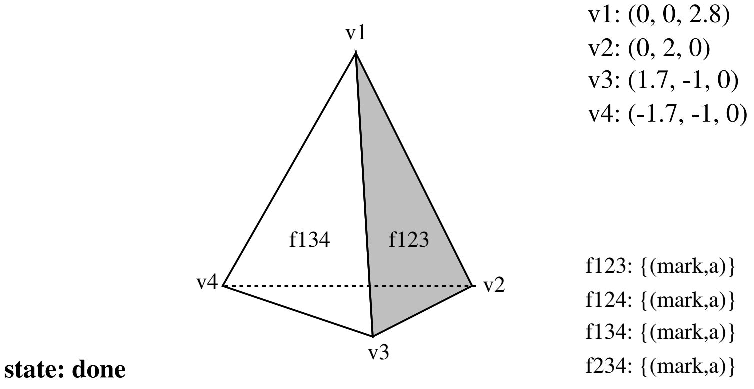


Figure 7.7: An initial solid.

For illustration purposes, we will use a tetrahedron as the initial solid of our example grammar. This tetrahedron, shown in Figure 7.7, is represented by the b-graph presented in Figure 7.6.

In a boundary solid grammar, rules match on portions of the initial solid and apply operations to modify the solid or generate additional solids. The next section introduces the formulation of solid rules and discusses how they are applied.

7.3 Reasoning About Solids

We express conditions or features of solids as clauses in first order logic. Explicit conditions of a given b-graph correspond to axioms about the boundary representation of a set of solids. Clauses (in the form of Horn clauses) allow deduction of complex conditions of the solids from simpler conditions. In this way, arbitrarily complex conditions may be specified using deductive reasoning on the solid representation. Locating a condition of a solid then becomes a matter of satisfying a goal clause that specifies the desired condition. An introduction to logic programming may be found in Sterling and Shapiro [50]. An account of the mathematical foundations of logic programming is presented by Lloyd [31].

Primitive Conditions

In order to express graph conditions as relations in first order logic, we present a mapping between the graph notation already presented and a relational notation, specified in Table 7.1. These relations are denoted *primitive conditions.* We use a logic programming notation [8] for these relations.

	Graph Notation	Relational Notation
Nodes	$k \in K_t$	$\texttt{t}(k)$
Arcs	$(k_1, k_2) \in \rho_a$	$\texttt{a}(k_1, k_2)$
Geometry	$\gamma(v) = c$	$\texttt{v\_coord}(v, c)$
Labels	$(a, v) \in \lambda(k)$	$\texttt{label}(k, a, v)$
State	$\sigma = s$	$\texttt{state}(s)$

Table 7.1: Conversion from graph notation to clausal notation.

Determining the type of topological elements may be represented as a relation $\texttt{t}(k)$, which denotes $k \in K_t$. For example, a vertex node $k \in K_{\text{VERTEX}}$ is denoted by the relation $\texttt{vertex}(k)$. Similarly, adjacency relations between topological elements $(k_1, k_2) \in \rho_a$ may be denoted by relations $\texttt{a}(k_1, k_2)$. Then the relation $\texttt{edgeh\_v}(e, v)$ denotes an $_{edgeh\_v}$ arc from e to v in ρ_{edgeh_v}. Coordinates of a vertex $\gamma(v) = c$ are denoted by the relation $\texttt{v\_coord}(v, c)$. For labels, the relation $\texttt{label}(k, a, v)$ denotes $(a, v) \in \lambda(k)$. The current state of a b-graph, $\sigma = s$, is denoted by the relation $\texttt{state}(s)$. Relations that directly relate topological element types and adjacency relations, coordinates, and labels are considered primitive conditions.

Satisfying primitive conditions of b-graphs is a straightforward task of locating nodes and arcs, and accessing their type, labeling, and geometric functions. These serve as axioms for deducing additional conditions.

For example, we can specify that there exists a specific edge-half e in a given b-graph by edge_half(e) which corresponds to finding a node $e \in K_{\text{EDGEHALF}}$. We can find the vertex v associated with that edge-half by the clause edgeh_v(e, v) This corresponds to locating an *edgeh_v*arc in the b-graph from e to v. The coordinates of the vertex can be accessed with the clause v_coord($v, (x, y, z)$) which corresponds to the geometric function γ, where $(x, y, z) = \gamma(v)$. A label l on an edge-half e can be found by specifying label(e, l) and locating l in the set of labels associated with that edge-half, $l \in \lambda(e)$. Finally, the state of a given b-graph may be found by specifying state(s).

Constructing Complex Conditions

Conditions may be combined using clauses in first order logic. A clause

$$A \leftarrow B_1, \ldots, B_n.$$

can be constructed to specify the conjunction of $B_1, \ldots, B_n$. Operationally, the satisfaction of A results as the sequential satisfaction of $B_1, \ldots, B_n$. For example, the implicit other_v relation, the relation between an edge-half and the vertex associated to its other edge-half, is a conjunction of the *other_eh*relation and the *edgeh_v*relation:

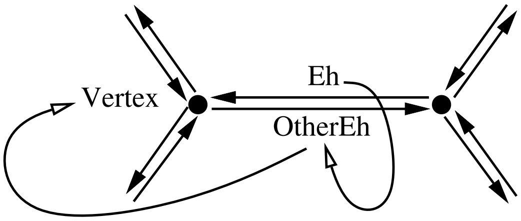


Figure 7.8: The implicit other_v relation.

```
other_v(Eh, Vertex):-
        other_eh(Eh, OtherEh),
        edgeh_v(OtherEh, Vertex).
```

Similarly, we express the length of an edge-half as the distance between the coordinates of the vertices associated with that edge-half and its other edge-half:

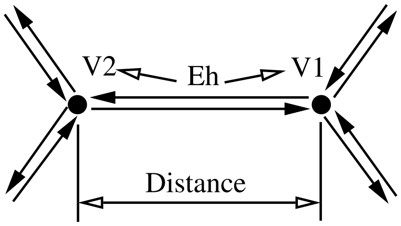


Figure 7.9: The length of an edge-half.

```
eh_length(Eh, Length):-
        edgeh_v(Eh, V1),
        other_v(Eh, V2),
        distance_v(V1, V2, Length).
```

where distance_v($V1, V2, Length$) calculates the Euclidean distance between the coordinates of two vertices $V1$ and $V2$.

In this way, diverse graph conditions and conditions on the geometry can be combined to form higher level conditions. Some of these conditions include:

- the *lengths* of edges (as demonstrated), *areas* of faces, *volumes* and *masses* of solids;
- *angles* between edges and faces;
- *orientations* of faces and solids;
- *coincident*, *colinear* or *coplanar* vertices, edges and faces;
- the *centers* of edges, faces and solids;
- the *moment of inertia* of faces and solids;
- *intersections* of lines, planes, surfaces, edges, faces, and solids, including the intersection of two solids (Boolean intersection) and *self-intersection* of a solid (unary intersection - described later in this thesis).

A single condition may be used to express the conditions of an infinite set of graphs using recursive definitions. Alternately, a condition may match greatly varied topology graphs using several clauses with the same head.

Composition and Decomposition of Elements

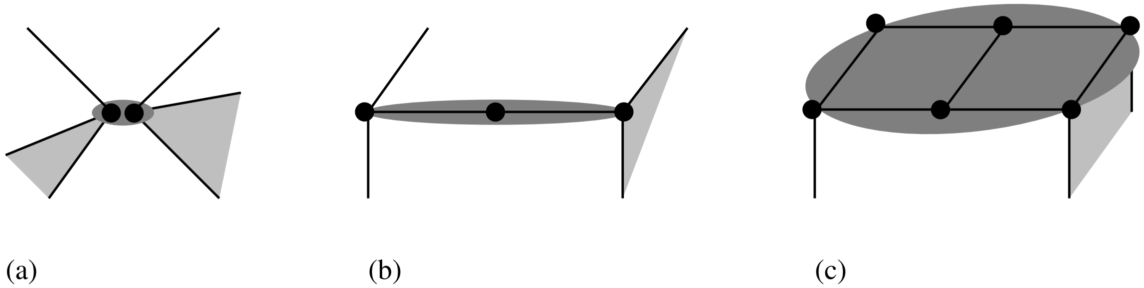


Figure 7.10: The decomposition of elements.

Locating features of solids is complicated by the composition and decomposition of elements, as described by Stiny [55]. We can illustrate the problem with the following example. Consider that we have a solid with one (straight) edge with a length of four units. We can then successfully locate that edge with the description given. However, we can split the edge into two edges (using `esplit`), and assign the coordinates of the new vertex to be the midpoint of the edge. We will still be representing the same physical solid, but we will not find an edge that is four units long. Instead, we have two adjacent edges, each two units long. We can have non-unique representations of the same object, and this complicates the description of our conditions.

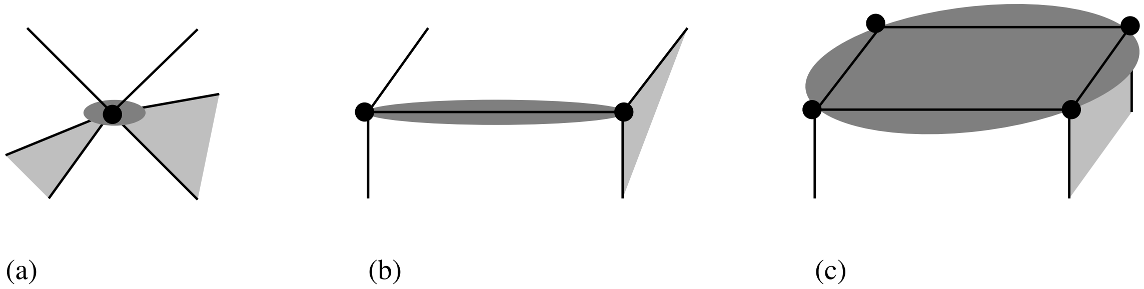


Figure 7.11: Composing elements.

This same problem occurs with elements of other dimensionalities. Coincident vertices, colinear edges, and coplanar faces are illustrated in Figure 7.10.

Two approaches may be used to address this problem. The first approach merges the elements. For example, the coincident vertices, colinear edges, and coplanar faces shown in in Figure 7.10 are shown merged in Figure 7.11 Unfortunately, these elements may be needed, so systematically reducing them may be counterproductive. For example, a portion of a face can be split off and labeled as a mating face of an assembly. Merging these faces would then incorrectly indicate the size and shape of the mating face.

The second approach uses flexible descriptions of conditions that allow for contiguous elements as if they were merged. It is straightforward to create and use an operator for this purpose. For example, the `non_colinear_eh` predicate used in the example rule in Chapter 7 and the generalized snowflake grammar in Chapter 8 traverses colinear edges as a single edge. The `non_colinear_eh` predicate finds the clockwise non-colinear edge half from the current one. The definition of the `non_colinear_eh` predicate is presented below.

```
cw_non_colinear_eh(Eh, CwNCEh):-
        cw_eh(Eh, CwEh),
        edgeh_v(Eh, V1),
        edgeh_v(CwEh, V2),
        v_coord(V1, C1),
        v_coord(V2, C2),
        cw_non_colinear_eh1(C1, C2, CwEh, CwNCEh).

cw_non_colinear_eh1(C1, C2, Eh, Eh):-
        other_v(Eh, V3),
        v_coord(V3, C3),
        not(colinear([C1, C2, C3])).

cw_non_colinear_eh1(C1, C2, Eh, CwNCEh):-
        cw_eh(Eh, CwEh),
        cw_non_colinear_eh1(C1, C2, CwEh, CwNCEh).
```

The `colinear` predicate determines if the points in a list are colinear.

7.4 Operations on Solids

Operations are used to transform one b-graph into another. *Primitive operations* modify the topology using Euler operations, modify the geometry using vertex coordinate assignment, add and remove labels, and modify the current state. Primitive operations are expressed as extra-logical relations, that directly modify the b-graph representation as a side-effect of being satisfied. Complex operations may be constructed from conditions and operations as clauses in first order logic. Complex operations may then produce a different effect depending on the context of their application.

Applying an operation to a solid then becomes a matter of satisfying a goal clause that specifies the desired operation.

Primitive Operations

Primitive operations have several forms. Euler operations modify boundary representation by adding and removing topological elements and relations, while maintaining valid topological adjacencies. They may be viewed as graph productions, and graph productions of several Euler operations are presented in Appendix A. Assigning coordinates c to a vertex v is equivalent to changing γ to γ' such that $\gamma'(v) = c$. Adding a label l to an element e is accomplished by modifying the labeling function λ to λ' such that $\lambda' = \lambda(e) \cup \{l\}$. Killing a label l modifies the labeling function λ to λ' such that $\lambda' = \lambda(e) - \{l\}$. Assigning coordinates c to a vertex v is equivalent to changing γ to γ' such that $\gamma'(v) = c$. modifying the current state of the b-graph is equivalent to modifying σ to σ' such that $\sigma' = s$. The Euler operations, coordinate assignment, labeling operations, and state modification comprise the primitive operations.

Euler operations are specified as extra-logical relations corresponding to the operations presented in Chapter 4. The relational form of these Euler operators are listed in Table 7.2.

```
mssflv(NewS,NewSh,NewF,NewL,NewV)
merge_solids(S1,S2)
msflv(S,NewSh,NewF,NewL,NewV)
mev(V,Eh,NewV,NewEh)
esplit(Eh,NewEh,NewV)
mefl(V1,PredEh,V2,SuccEh,NewEh,NewL,NewF)
keml(Eh,NewL)
glue(F1,F2)
kssflevs(S)
ksflevs(Sh)
kev(Eh)
ejoin(Eh)
esqueeze(Eh)
kefl(Eh)
mekl(V1,PredEh,V2,SuccEh,NewEh)
unglue(CycleOfEhs,NewF1,NewF2)
ksv(V1,V2)
kvmg(V1,V2)
keg(Eh1,Eh2)
msv(V1,V2)
mvkg(V1,V2)
meg(Eh1,Eh2)
```

Table 7.2: Euler operations as primitive operations.

For example,

```
mev(V,Eh,NewV,NewEh)
```

is equivalent to the `mev` Euler operation shown in Figure 4.5.

The inversion operation, described in Chapter 5, is specified as

`invert(S,NewS).`

The generalized unary intersection is specified as

`unary(N,S,NewS).`

Vertex coordinates are assigned using

`set_vertex`(v, c),

which is equivalent to modifying γ to γ' such that $\gamma'(v) = c$. Element labels are created using

`make_label`(e, a, v),

which is equivalent to changing λ to λ' such that $\lambda'(e) = \lambda(e) \cup \{(a, v)\}$, or

`kill_label`(e, a, v)

which modifies λ to λ' such that $\lambda'(e) = \lambda(e) - \{(a, v)\}$. The current state of the b-graph is modified with

`set_state`(s).

which is equivalent to modifying σ to σ' such that $\sigma' = s$.

Primitive operations derive one b-graph from another according to the following definition:

Definition 7.5 *A b-graph $b' \in b(\Sigma_\star)$ is* **directly derivable** *from another $b \in b(\Sigma_\star)$ by a primitive operator O (abbreviated $b \underset{O}{\Longrightarrow} b'$) iff*

1. *the arguments to O have valid types and adjacencies for that particular operation;*
2. *O applied to b produces b'.*

Constructing Complex Operations

Complex operations are specified by sequences of conditions and operations. Primitive operations modify the topology, geometry, labels, and state of a b-graph. Conditions, as described above, are used to control the use of operations in modifying the given b-graph. Clauses define high level operators from existing operators and conditions. The conditions allow an operator to respond to the context in which it is applied.

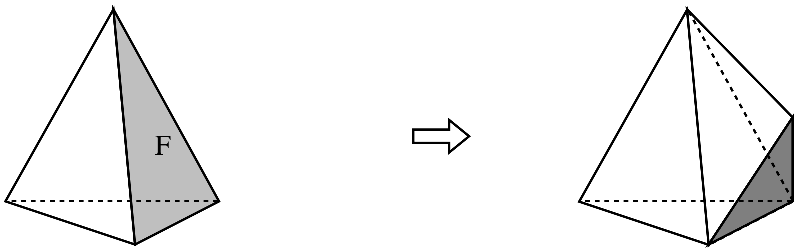


Figure 7.12: The point_face operator.

Conditions and operations may be combined using clauses in first order logic. A clause

$$A \leftarrow B_1, \ldots, B_n.$$

can be constructed to specify the conjunction of $B_1, \ldots, B_n$. Operationally, the satisfaction of A results as the sequential satisfaction of $B_1, \ldots, B_n$.

For example, the `point_face` operator, illustrated in Figure 7.12, will pull any face out into a point, correctly modifying the topology while considering the number of edges and vertices of the face. The clauses for the `point_face` operator are listed below.

```
point_face(Face, Height):-
        face_eh(Face, Eh),
        ccw_eh(Eh, LastEh),
        edgeh_v(Eh, V),
        face_normal(Face, Normal),
        face_center(Face, Center),
        mev(V, LastEh, VTop, EhBt),
        other_eh(EhBt, EhTb),
        scalar(Height, Normal, Direct),
        vecplus(Center, Direct, CTop),
        set_vertex(VTop, CTop),
        point_face_1(LastEh, Eh, VTop, EhTb).

point_face_1(EndEh, EndEh, _, _).
point_face_1(EndEh, Eh, VTop, EhTb):-
        cw_eh(Eh, NextEh),
        edgeh_v(NextEh, V),
        v_coord(V, C),
        mefl(V, Eh, VTop, EhTb, NewEhBt, _, _),
        other_eh(NewEhBt, NewEhTb),
        point_face_1(EndEh, NextEh, VTop, NewEhTb).
```

General operators, such as offsetting, Boolean, and unary shape operators, may be constructed in this way. In addition, we can construct a single operation to be used in place of a set of rules.

7.5 The Composition of Solid Rules

A solid rule is specified by a set of match conditions, and a sequence of operations. The match conditions of a rule determine when the rule may be applied to a given b-graph. Each of the rule's conditions must be satisfied with respect to the given b-graph, and bindings for all the free variables must be found. When the rule is applied, the sequence of operations transform the b-graph, modifying the representation of the solid(s) or creating additional solids.

Definition 7.6 *A solid rule $\mathcal{R}$ is a pair $\alpha \Rightarrow \beta$, where*

- *α is a goal clause $\leftarrow C_1, \ldots, C_n$, where each condition C_i is a subgoal that must be satisfied in order to apply the rule; and*
- *β is a goal clause $\leftarrow O_1, \ldots, O_n$, where each operation O_i is considered a subgoal. If the operations $O_1, \ldots, O_n$ of β are not satisfied with respect to the given b-graph then β does not modify the b-graph (i.e. β is equivalent to the identity transformation).*

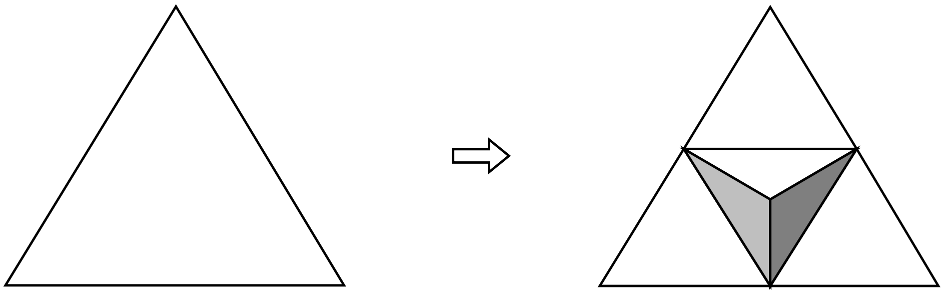


Figure 7.13: A simple solid rule.

We can clarify the idea of a solid rule with a simple example.

```
description(point_1, 'Modify a face.').

lhs(point_1, [F123], [F123]):-
        face(F123).

rhs(point_1, [F123]):-
        face_eh(F123, FirstEh),
        cw_non_colinear_eh(FirstEh, Eh12),
                cw_non_colinear_eh(Eh12, Eh23),
                eh_distance(Eh12, Eh23, Length),
                face_midpoint_esplit(F123),
                point_face(F123, Length/2.449489743).
```

The left hand side (lhs) of the rule matches on any face of a b-graph. The matched face will bind to the variable F123. The first argument of `lhs` is the name of the solid rule, the second argument is a list of common variables between the `lhs` and `rhs`, and the third argument is a list of topological elements that will be highlighted in the graphical display of the solids when the rule matches.

When the rule is applied, the operations and matching of the right hand side (rhs) will be executed in sequence. `face_eh` will find the first edge-half of the face bound to F123 and bind it to the variable FirstEh. `cw_non_colinear_eh` finds the first edge-half that is clockwise about the loop of edges of the face, and is not colinear with the given edge-half. This condition is applied twice to find a maximal edge of the face. `eh_distance` is used to calculate the length of this maximal edge. `face_midpoint_esplit` is an operation that splits each of the maximal edges of a face at the midpoint, if a vertex does not already exist at that point. Finally, `point_face` splits the given face, and pulls out the newly created vertex by the given amount, as shown in Figure 7.12. The constant 2.449489743 is an approximation of the square root of 6. The application of the rule to a triangular face is illustrated in Figure 7.13.

Solid rules derive one b-graph from another according to the following definition:

Definition 7.7 *A b-graph $b' \in b(\Sigma_\star)$ is* **directly derivable** *from another $b \in b(\Sigma_\star)$ by a solid rule $\mathcal{R}$ (abbreviated $b \underset{\mathcal{R}}{\Longrightarrow} b'$) iff*

1. *the goal $\alpha_\mathcal{R}$ is satisfiable with respect to the primitive conditions derivable from b and the clauses in P_C;*
2. *one of two cases holds:*
   (a) *the goal $\beta_\mathcal{R}$ is satisfiable with respect to the primitive conditions derivable from b and subsequent b-graphs as modified by the operations in $\beta_\mathcal{R}$, and with respect to the clauses in P_C and P_O; and b' is directly derivable from b by the sequence of primitive operations applied in satisfying $\beta_\mathcal{R}$; or*
   (b) *the goal $\beta_\mathcal{R}$ is unsatisfiable and $b' = b$.*

Match Complexity

The complexity of rule matching in boundary solid grammars is specific to the underlying representation and access procedures. The complexity of each of the primary access methods for the

Genesis boundary solid grammar interpreter (see Chapter 9) is listed in Table 7.3, where l_e is the number of labels on a specific element, and l_t is the total number of labels. These complexities are based on analyses of the algorithms used to implement the access routines.

item to find	*complexity*
any element	$O(1)$
a specific element	$O(1)$
the type of an element	$O(1)$
an element adjacent to a specific element	$O(1)$
the coordinates of a vertex	$O(1)$
the equation of a face	$O(1)$
the bounding box of a face	$O(1)$
the bounding box of a solid	$O(1)$
any label	$O(1)$
a specific label on specific element	$O(l_e)$
a specific label on any element	$O(l_t)$

Table 7.3: Access complexity of various elements

Conditions can have a wide range of complexity. Simple conditions may be satisfied in constant time complexity. For example, the sample solid rule given above may be applied to any face. The match complexity of that rule would depend only on the complexity of the access routine to find any face, $O(1)$. Match conditions may have exponential or factorial time complexity.

To demonstrate a NP-complete match condition, we will use a decision problem related to the traveling salesman problem [19]:

Traveling Salesman
Instance: A finite set $V = \{v_1, v_2, ..., v_m\}$ of vertices, a distance $d(v_i, v_j) \in \Re^+$ for each pair of vertices $v_i, v_j \in V$, and a bound $B \in \Re^+$ (where $\Re^+$ denotes the positive real numbers).
Question: Is there a tour of all the vertices in V having total length no more than B, that is, an ordering $\langle v_{\pi(1)}, v_{\pi(2)}, ..., v_{\pi(m)} \rangle$ of V such that $B \geq d(v_{\pi(m)}, v_{\pi(1)}) + \sum_{i=1}^{m-1} d(v_{\pi(i)}, v_{\pi(i+1)})$?

This translates to a match condition with the following clauses:

```
traveling_salesman(B, D, VertexList):-
        vertex(V),
        B > 0,
        traveling_salesman_1(B, D, VertexList, 0, [V]),
        not(vertex_not_used(VertexList)).

traveling_salesman_1(B, D, VertexList, Dlast, [Vlast | Rest]):-
```

```
        vertex(V),
        not(member(V, [Vlast | Rest])),
        distance_v(V, Vlast, Dadd),
        Dnew = Dadd + Dlast,
        B >= Dnew,
        traveling_salesman_1(B, D, VertexList, Dnew, [V, Vlast | Rest]).

traveling_salesman_1(B, D, VertexList, D, VertexList).

vertex_not_used(VertexList):-
        vertex(V),
        not(member(V, VertexList)).
```

This problem is of course NP-complete. We will not want to use this match for models with large numbers of vertices, however it is useful to demonstrate the descriptive power of the formalism and the complexity of possible matches.

Fortunately, all the grammars investigated so far have rules with low match complexities.

7.6 Boundary Solid Grammars

With the representation, initial solids, and solid rule definitions in hand, we can now define boundary solid grammars.

Definition 7.8 *A* **boundary solid grammar** *is a tuple* $\mathcal{G} = (\Sigma_\star, I, P, R)$, *where*

1. $\Sigma_\star$ *is as defined for b-graphs;*
2. $I \in b(\Sigma_\star) \cup \{b_\varepsilon\}$ *is a topologically valid initial b-graph;*
3. $P = P_C \cup P_O$ *is a finite set of clauses, where* P_C *is a set of clauses specifying conditions, and* P_O *is a set of clauses specifying operations; and*
4. R *is a finite set of solid rules.*

The initial solid is a topologically valid solid or collection of solids in the above representation. Modifications of the initial solid, and subsequent solids, are accomplished by the application of the set of solid rules.

We can abbreviate $\mathcal{G} = (\Sigma_\star, I, P, R)$ to $\mathcal{G} = (I, P, R)$ since Σ_{node}, Σ_{arc}, $\Re^3$ (of $\Sigma_\star$) are fixed for a given boundary representation, and Σ_{label} may be a fixed alphabet of labels.

7.7 Languages of Solids

The sentential set $S(\mathcal{G})$ of a boundary solid grammar $\mathcal{G}$ is the set of b-graphs (sentential b-graphs) which contains the initial b-graph and all b-graphs which can be generated from the initial b-graph using the solid rules of the grammar.

A boundary solid grammar $\mathcal{G}$ generates a language $L(\mathcal{G})$ of b-graphs. The language of a boundary solid grammar is the set of sentential b-graphs where the current state is **done**, i.e., $L(\mathcal{G}) = \{b \in S(\mathcal{G}) \mid \sigma(b) = \textbf{done}\}$.

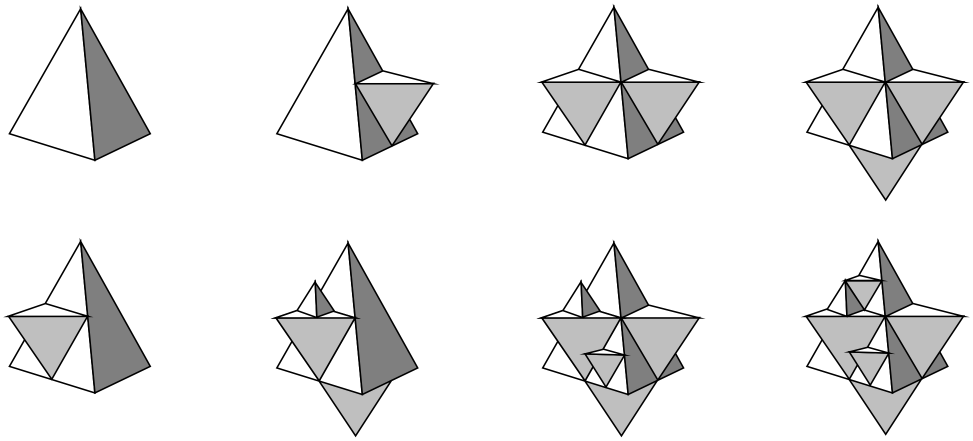


Figure 7.14: A portion of the language of snowflakes.

The example initial solid and solid rule define a boundary solid grammar. A language of snowflakes or crystal forms is produced by this solid grammar that includes both regular and irregular examples. A portion of this language is shown in Figure 7.14. A more detailed snowflake, shown in Figure 7.15, was produced by the **Genesis** boundary solid grammar interpreter. It is interesting to note that that the members of the language of this grammar may have very complex forms, and that there is an infinite number of members in this language. However, the description of the grammar is quite concise.

The boundary solid grammar formalism is intended to provide a mechanism for generating languages that satisfy some design criteria, specified as constructive rules of design.

Figure 7.15: Another snowflake.

Chapter 8

Applications of Boundary Solid Grammars

Several grammars have been developed that demonstrate the ideas and utility of boundary solid grammars. These grammars are presented in this chapter.

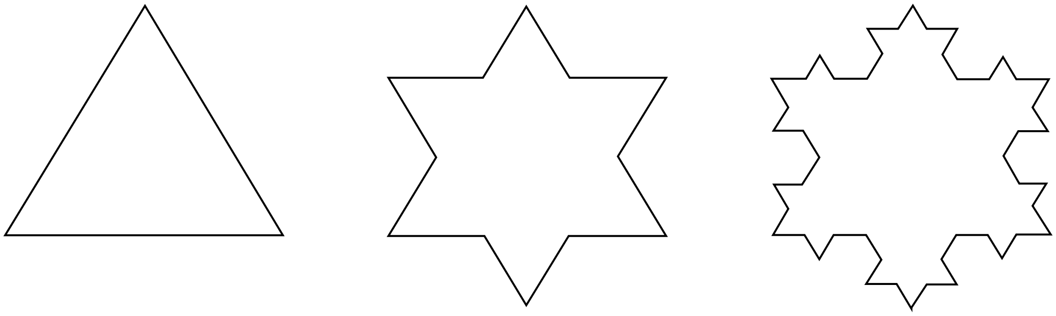


Figure 8.1: Three stages of generation of the Koch snowflake.

The first boundary solid grammar that was constructed generates a three dimensional variant of the Koch snowflake. The Koch snowflake, shown in Figure 8.1, was introduced in 1904 by Helga von Koch as an example of a curve of infinite length that contains a finite area. The 3-D snowflake, illustrated in Figure 8.2, has an infinite surface area and contains a finite volume. We have developed snowflake grammars that generate both uniform and non-uniform snowflakes. An associated grammar generates uniform subdivisions of octahedra. Solid models of mountains are generated by a third grammar (a variant of the uniform snowflake grammar). A large variety of conch-like shells are generated by a spiral grammar. Another grammar automatically generates supports for models to be made by a stereo lithography (sla) process. The final grammar presented here generates 3-D designs for Queen Anne houses.

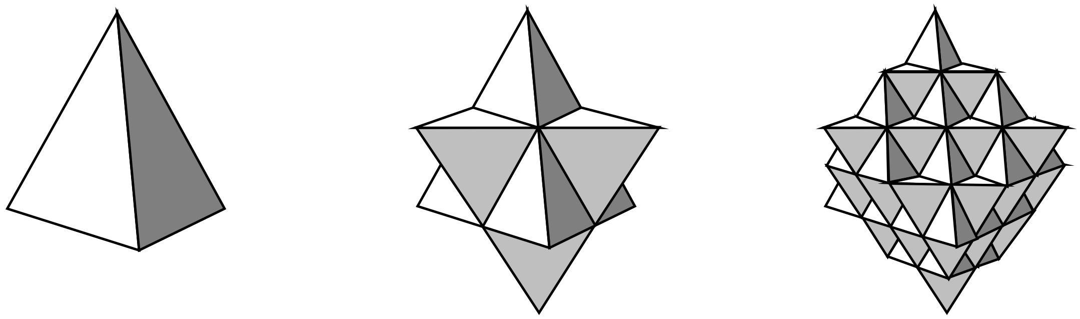


Figure 8.2: Three stages of generation of a 3-D snowflake.

At the time of writing, two additional grammars are under development. One grammar constructs computer housings for single board computers for the MICON project. The second grammar generates structural designs of high rise buildings.

8.1 Generalized Snowflake Grammars

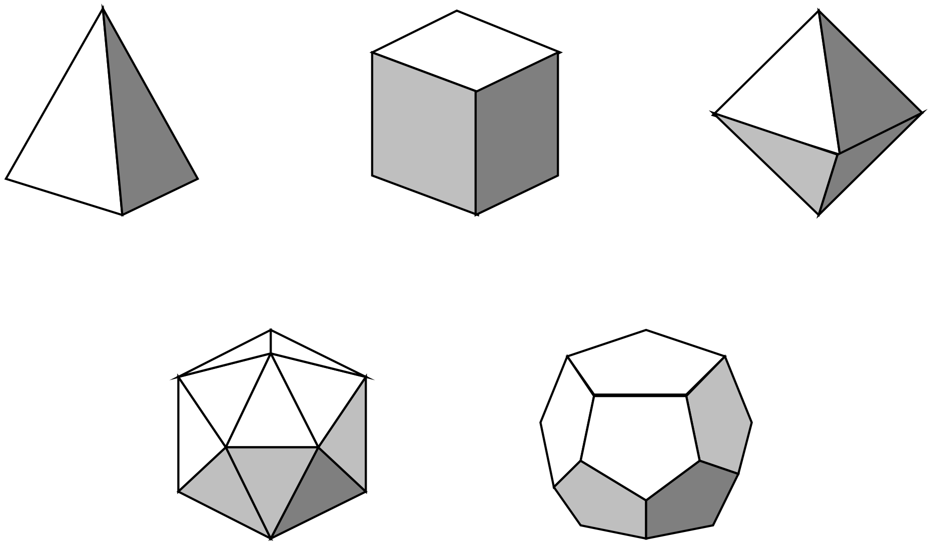


Figure 8.3: Alternative initial solids.

The rule presented as an example in Chapter 7 is much more general than it might first appear. (A listing of the rule is repeated below.)

```
description(point_1, 'Build a point on a face.').

lhs(point_1, [F123], [F123]):-
```

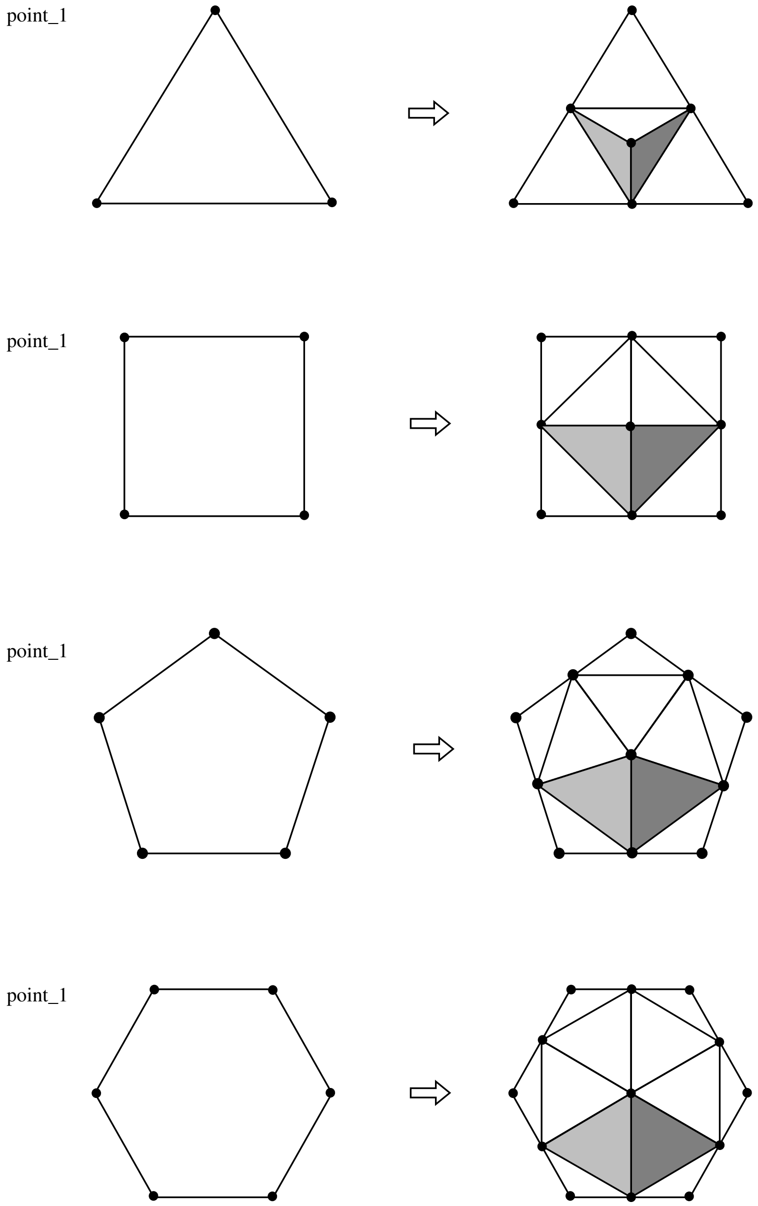


Figure 8.4: The generalized snowflake rule applied to various faces.

```
        face(F123).

rhs(point_1, [F123]):-
        face_eh(F123, FirstEh),
        cw_non_colinear_eh(FirstEh, Eh12),
        cw_non_colinear_eh(Eh12, Eh23),
```

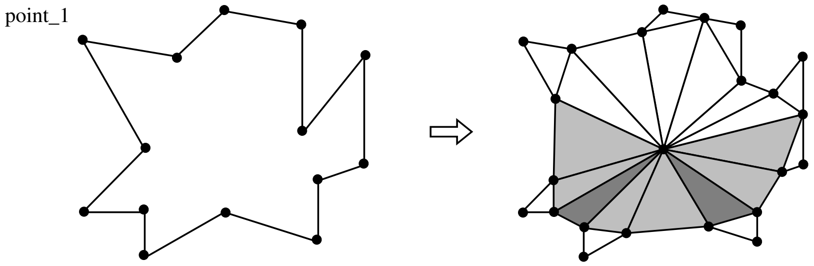


Figure 8.5: The generalized snowflake rule applied to a concave face.

```
eh_distance(Eh12, Eh23, Length),
face_midpoint_esplit(F123),
point_face(F123, Length/2.44949).
```

In addition to the triangular faces illustrated, Rule `point_1` matches on any face regardless of the number of edges and vertices, or if it is convex. The application of the rule to faces with different numbers of edges and vertices is shown in Figure 8.4. The application of the rule to a concave face is shown in Figure 8.5.

We can vary the grammar by using different initial solids. The original grammar used the tetrahedron in Figure 7.7 as its initial solid. Since the rule can match on any face, we can construct grammars using any solid as the initial solid. For example, we could use any of the Platonic solids, as in Figure 8.3. In this way, we can create families of grammars with a common rule set.

8.2 Uniform Snowflake Grammars

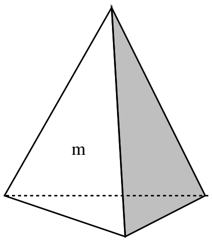


Figure 8.6: An initial solid for the uniform snowflake grammars.

If we want the language of the snowflake grammar to include only uniform snowflakes, as in Figure 8.2, then we need to modify the grammar. We can generate uniform snowflakes using four variations of the rule from the previous snowflake grammar, and the addition of a label to the initial solid. The new initial solid is shown in Figure 8.6, and the new rules are presented in Figure 8.7.

The definition of `snowflake_2` from Figure 8.7 is listed below.

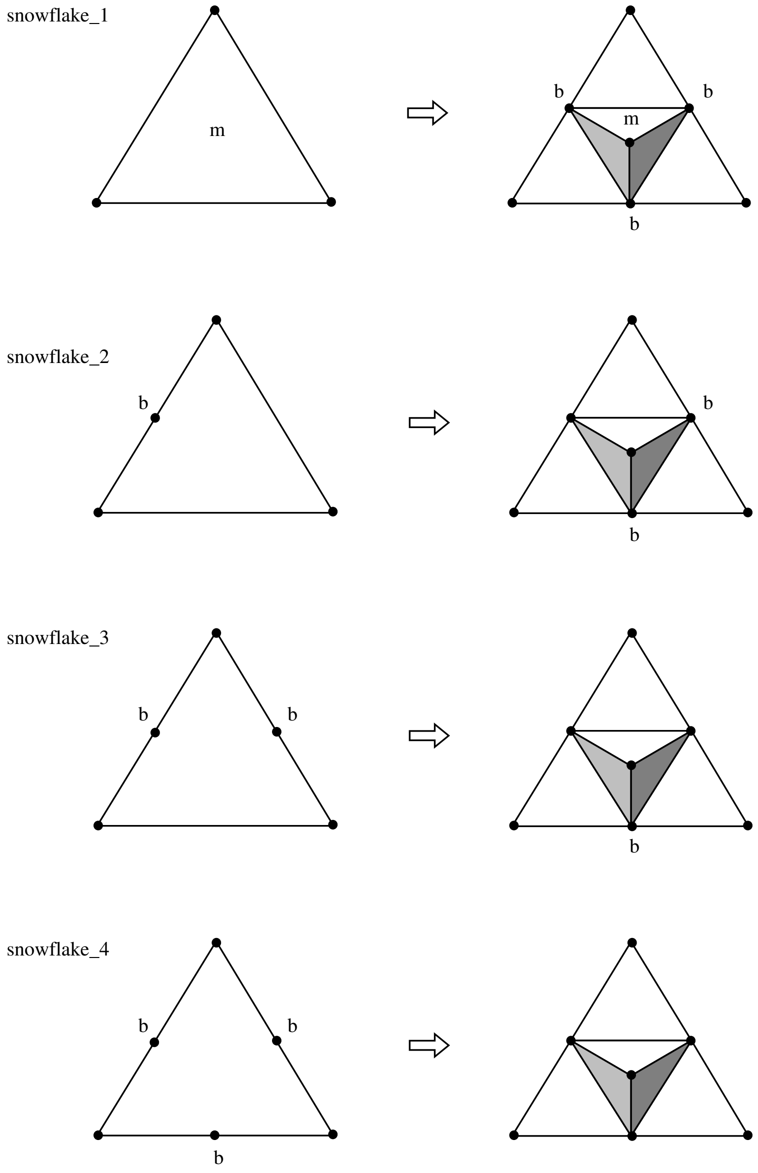


Figure 8.7: Rules for the uniform snowflake grammars.

```
description(snowflake_2, 'Modify an inner (one B) triangular face.').

lhs(snowflake_2, [F123, Eh12,Eh25,Eh53,Eh31, V5], [F123]):-
        state(tetra2),
        label(V5, mark, b),
        edgeh_v(Eh53, V5),
        cw_eh(Eh53, Eh31),
```

Figure 8.8: A uniform snowflake.

```
        cw_eh(Eh31, Eh12),
        cw_eh(Eh12, Eh25),
        cw_eh(Eh25, Eh53),
        edgeh_f(Eh12, F123).

rhs(snowflake_2, [F123, Eh12,Eh25,Eh53,Eh31, V5]):-
        face_sh(F123, Shell),
        esplit_midpoint(Shell,Eh12,Eh42,V4),
        esplit_midpoint(Shell,Eh31,Eh61,V6),
        mefl(Shell,V6,Eh31,V4,Eh42,Eh64,L146,F146),
        mefl(Shell,V4,Eh64,V5,Eh53,Eh45,L254,F254),
        mefl(Shell,V5,Eh45,V6,Eh64,Eh56,L365,F365),
        eh_length(Eh12, Length),
        point_face(F123, Length/1.2247449),
        kill_label(V5, mark, b),
        make_label(V4, mark, b),
        make_label(V6, mark, b).
```

Rule snowflake_2 modifies a triangular face (with 4 vertices) that is labeled with "(mark,a)" edge labels and "(mark,b)" vertex labels, dividing it at the midpoints of the original three edges, and relabeling it with new "(mark,a)" labels. (The figures abbreviate labels of the form "(mark,a)" to “a”.) The constant 2.44949 is approximately the square root of 6, and 1.2247449 is half of the square root of 6.

The four rules of the new grammar propagate labels around the snowflake starting from a single face of the initial solid.

8.3 Uniform Octahedron Grammar

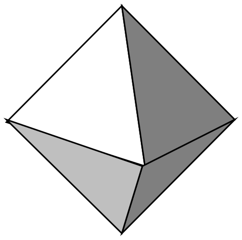


Figure 8.9: Initial solid for the uniform octahedron grammar.

Figure 8.10: A uniform octahedron.

The octahedron grammar generates solids that are recursive assemblies of octahedra. However, they are created by recursive subdivision of a initial octahedron. The patterns of indentations on the sides of the octahedron correspond to the Sierpinski gasket [32].

The grammar uses another variation of the uniform snowflake grammar. It starts begins with an octahedron as the initial solid, as shown in Figure 8.9. Instead of pulling out a face, the octahedron rules pushes an indentation into the face. The rules for the uniform octahedron grammar are similar to those illustrated in Figure 8.7.

8.4 Mountain Grammars

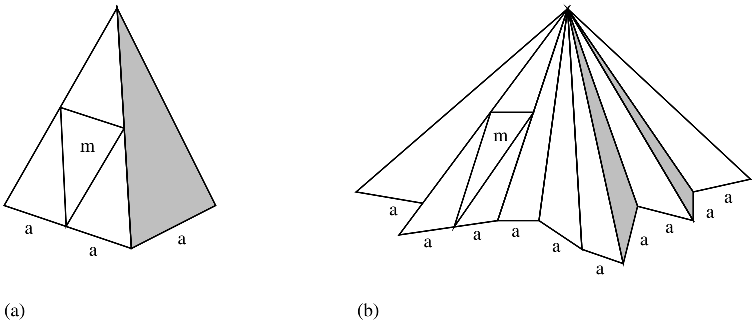


Figure 8.11: Initial solids for the uniform mountain grammars.

The "fractal" mountain grammars follow directly from the uniform snowflake grammars already presented. They assign random geometry to the vertices of the faces modified, and limit the application of rules to within marked boundaries (to keep the mountains from growing into the adjacent "foothills").

The first of these two changes requires a minor modification to the four rules of the snowflake grammar. The second requires four additional rules to handle the boundary conditions. Figure 8.12 and Figure 8.13 illustrate the rules of this grammar.

Rule mount_2 in Figure 8.12 modifies a triangular face (with 4 vertices) that is labeled with "(mark, a)" edge labels and "(mark, b)" vertex labels, dividing it at the midpoints of the original three edges, and relabeling it with new "(mark, a)" labels.

```
description_mount(mount_2, 'Modify an inner (one B) triangular face.').

lhs_mount(mount_2, [Eh12,Eh25,Eh53,Eh31, V5],[F123]):-
        state(mount),
        label(V5, mark, b),
        edgeh_v(Eh53, V5),
        cw_eh(Eh53, Eh31),
        no_label(Eh31, mark, a),
        cw_eh(Eh31, Eh12),
        no_label(Eh12, mark, a),
```

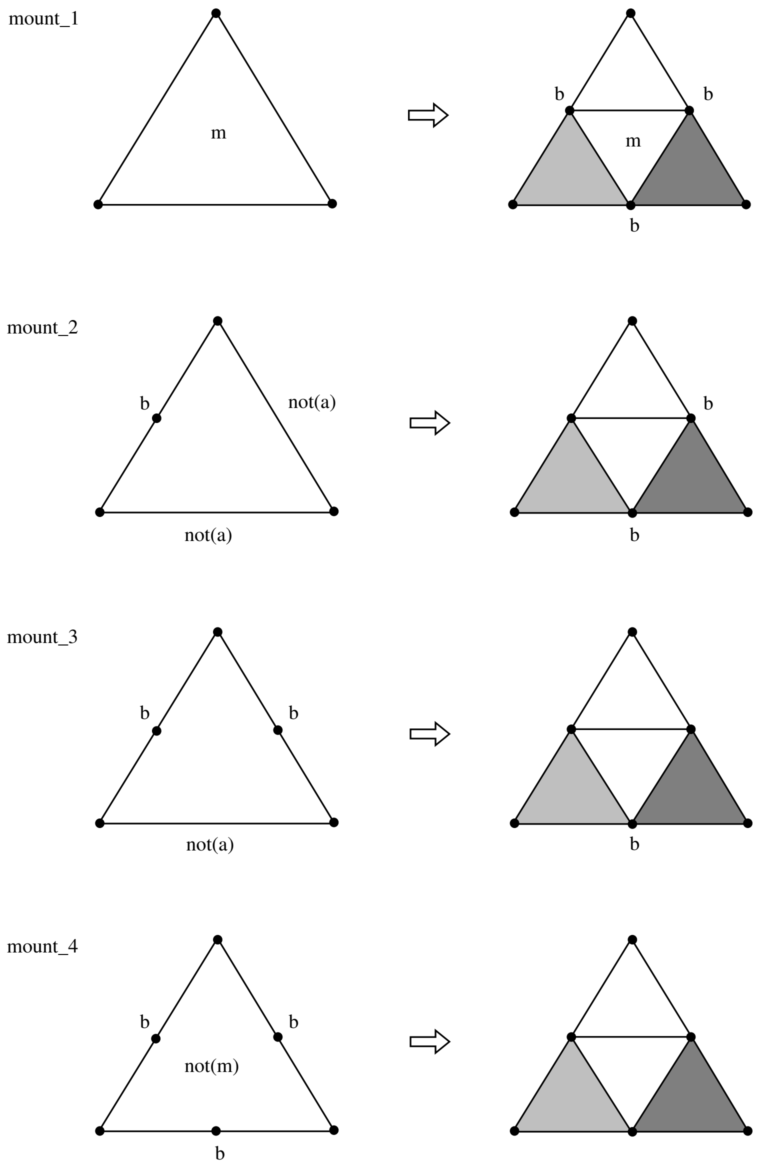


Figure 8.12: Basic rules for the uniform mountain grammars.

```
        cw_eh(Eh12, Eh25),
        cw_eh(Eh25, Eh53),
        edgeh_f(Eh53, F123).

rhs_mount(mount_2, [Eh12,Eh25,Eh53,Eh31, V5]):-
        esplit_midpoint(Eh12,Eh42,V4),
        esplit_midpoint(Eh31,Eh61,V6),
```

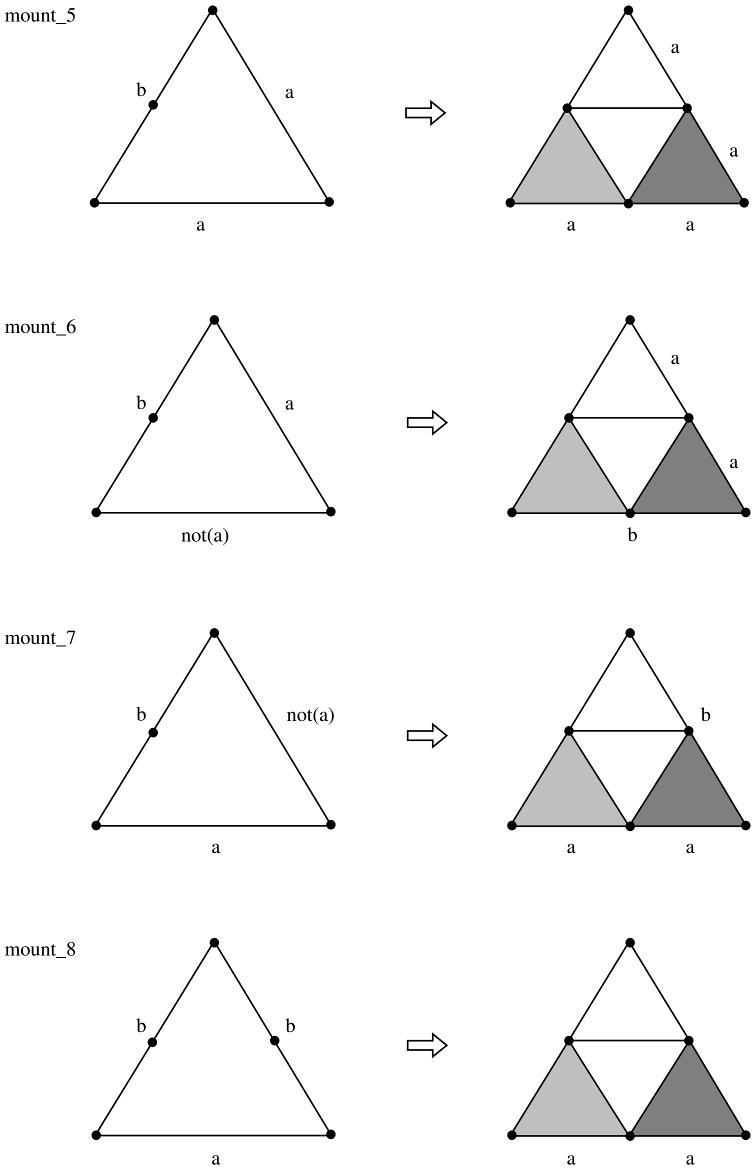


Figure 8.13: Border rules for the uniform mountain grammars.

```
mefl(V6,Eh31,V4,Eh42,Eh64,L146,F146),
mefl(V4,Eh64,V5,Eh53,Eh45,L254,F254),
mefl(V5,Eh45,V6,Eh64,Eh56,L365,F365),
move_vertex_random_eh(Eh53),
kill_label(V5, mark, b),
make_label(V4, mark, b),
make_label(V6, mark, b).
```

8.5 Spiral Grammars

The spiral grammar illustrates the flexibility of matching and operations in boundary solid grammars. The single rule of this grammar will apply to *any* face, without regard for the number of loops, edges, or vertices. However in making spiral forms, the rule is restricted to apply only to a face with a labeled edge-half).

```
description(spiral_1, 'Add growth to the spiral.').

lhs(spiral_1, [StartEh], [Face]):-
        state(go_spiral),
        label(StartEh, mark, start),
        edgeh_f(StartEh, Face).

rhs(spiral_1, [StartEh]):-
        offset_factor(Ofactor),
        scale_matrix(Smatrix),
        rotate_matrix(Rmatrix),
        eh_length(StartEh, RefDist),
        edgeh_f(StartEh, Face),
        face_center(Face, Center),
        face_normal(Face, Normal),
        Dist = Ofactor * RefDist,
        scalar(Dist, Normal, Ndist),
        mattranslate(Ndist, Nmatrix),
        mattranslate(Center, Cmatrix),
        matinverse(Cmatrix, InvCmatrix),
        matmult([Cmatrix, Rmatrix, Smatrix, InvCmatrix, Nmatrix], Matrix),
        extrude_face_nonplanar_eh(StartEh, Matrix, NewEh),
        kill_label(StartEh, mark, start),
        make_label(NewEh, mark, start).
```

The application of spiral_1 rule is illustrated in Figure 8.16.

8.6 Hunting The Spiny Icosopus

New grammars may be formed by combining existing grammars. This section illustrates the idea of combining grammars with an example derivation, the "spiny icosopus". Our spiny icosopus is generated using the spiral, uniform snowflake, and mountain grammars already presented in this chapter.

We begin with an icosahedron “egg” of the spiny icosopus as the initial solid, as shown in Figure 8.18. The spiral grammar is used to generate spiral “arms” from each of the faces of the icosahedron, resulting in the model shown in Figure 8.19. This model is used as the initial solid of the uniform snowflake grammar. The rules of the uniform snowflake grammar generate the tetrahedra “spines”, as shown in Figure 8.20. The resulting model is used as the initial solid of the mountain grammar, which gives some irregularity to the surface of the icosopus. The completed spiny icosopus is illustrated in Figure 8.21.

The possibilities of combining these grammars in different ways is virtually limitless. We could use different initial solids, apply the grammars in different orders, and use different derivations, to generate an endless stream of distinct solids.

The grammars and geometric forms discussed thus far in this chapter are of academic interest. The next two sections discuss applications of boundary solid grammars that address existing modeling problems.

8.7 Stereo Lithography Support Grammars

A stereo lithography apparatus (SLA) allows the construction of epoxy prototypes directly from a solid model. Prototypes are produced by hardening epoxy resin with a computer directed laser. This process is accomplished a layer at a time, working from the lowest layer on the model upward. In order to support portions of the model that are not connected to the lowest layer (and the platform that supports it) support structures are needed.

The stereo lithography support grammars generate structures to support a model given as the initial solid of the grammar. Two rules of this grammar are presented below.

`sla_2` locates edges that will need support during the stereo lithography process. When the rule is applied, the `make_edge_support` operation is invoked to generate an edge support structure.

```
description(sla_2, 'Support an edge of a solid.').

lhs(sla_2, [Eh], [Eh]):-
        state(go),
        label(S, sla, original),
        solid(S),
        face_solid(F, S),
        face_normal(F, [_,_,Z]),
        Z < -0.5,
        not(label(F, sla, supported_f)),
        edgeh_f(Eh, F),
        not(label(Eh, sla, supported_e)),
        other_eh(Eh, Oeh),
                not(label(Oeh, sla, supported_e)),
                min_unsupported_span(Span),
                eh_length(Eh, Length),
                Length > Span.

        rhs(sla_2, [Eh]):-
                make_edge_support(Eh),
                make_label(Eh, sla, supported_e).
```

sla_3 locates vertices that are not supported, and generates a vertex support structure.

```
        description(sla_3, 'Support a vertex of a solid.').

        lhs(sla_3, [V], [V]):-
                state(go),
                label(S, sla, original),
                solid(S),
                face_solid(F, S),
                face_normal(F, [_,_,Z]),
                Z < -0.5,
                vertex_f(V, F),
                not(label(V, sla, supported_v)).

        rhs(sla_3, [V]):-
                make_vertex_support(V),
                make_label(V, sla, supported_v).
```

Applications of Rule sla_2 and Rule sla_3 are illustrated in Figure 8.22. A model with support structures generated by **Genesis** is shown in Figure 8.23.

8.8 Queen Anne Grammar

The grammar for Queen Anne houses is based on Flemming's shape grammar [18, 17]. Queen Anne houses are generated with rules for layout, room naming, roofs, room articulation, and porches. Building details are generated by additional rules, including interior walls, porch columns, window layout and details, chimney placement and construction, and roof dormers. The current grammar contains 136 rules. Based on a conservative combinatorial analysis, the Queen Anne grammar generates more than 5 trillion different houses. Examples of Queen Anne houses generated by the grammar are presented in Figures 1.1, 8.28, and 8.29. A sample of rules of the Queen Anne grammar is presented below.

Rule qa_1 adds a room by extending a plan towards the side or back. It looks for a side face of the hall which is not the front face, and checks if there are no more than 2 rooms at the side or back. When qa_1 is applied, it adds a room to that (back, left, or right) face of the hall, and transfers labels to the right places. The application of qa_1 is illustrated in Figure 8.24.

```
description(qa_1, 'Add a room adjacent to the hall.').

lhs(qa_1, [Eh, Depth, Vl, Vr, Cl, Chl, Chr, Cr, Xl, Xr], [Side]):-
        state(rooms),
        label(Hall, name, hall),
        bottom(Floor, Hall),
        side_not_front(Side, Hall),
        edgeh_f(Eh, Side),
        othereh_f(Eh, Floor),
        edgeh_v(Eh, Vhl),
        other_v(Eh, Vhr),
        v_coord(Vhl, Chl),
        v_coord(Vhr, Chr),
        label(Vl, mark, Xl), member(Xl, [back, front]),
        label(Vr, mark, Xr), member(Xr, [back, front]),
        not_same(Vl, Vr),
        member_once(back, [Xl, Xr]),
        v_coord(Vl, Cl),
        v_coord(Vr, Cr),
        colinear([Cl, Chl, Chr, Cr]),
        ordered([Cl, Chl, Chr, Cr]),
        distance(Cl, Cr, Distance),
        other_eh(Eh, OEh),
        room_shorter_width(Hall, Width, OEh),
        room_depth(Depth),
        Distance < Depth*2 + Width.


rhs(qa_1, [Eh, Depth, Vl, Vr, Cl, Chl, Chr, Cr, Xl, Xr]):-
        stack_solid(Eh, Depth, Room, [_, _, _, Right, _, Left],
                                     [_, V2, _, V3 | _ ]),
        distance(Chl, Cl, Left_pull),
        distance(Chr, Cr, Right_pull),
        pull_face(Left,  Left_pull),
        pull_face(Right, Right_pull),
        kill_label(Vl, mark, Xl),
        kill_label(Vr, mark, Xr),
        make_label(V2, mark, Xl),
        make_label(V3, mark, Xr),
        make_label(Room, name, room),
        room_color(RColor),
        set_solid_color(Room, RColor).
```

qa_6 designates a front room as a parlor. It looks for a room at the front of the house (but not the hall), and marks the room as parlor. This is illustrated in Figure 8.25.

```
description(qa_6, 'Locate the parlor to the front of the house.').

lhs(qa_6, [Parlor], [Parlor]):-
        state(rooms),
        label(Parlor, name, room),
        vertex_solid(V, Parlor),
        label(V, mark, front).

rhs(qa_6, [Parlor]):-
        set_state(parlor),
        kill_label(Parlor, name, room),
        make_label(Parlor, name, parlor),
        parlor_color(ParColor),
        set_solid_color(Parlor, ParColor).
```

Rule qa_11 selects a room to be the dining room. The rule matches on an unnamed room which is adjacent to the kitchen, and marks the room as a dining room. The application of qa_11 is shown in Figure 8.26.

```
description(qa_11, 'Locate the dining room next to the kitchen.').

lhs(qa_11, [Dining], [Dining]):-
        state(kitchen),
        label(Kitchen, name, kitchen),
        label(Dining, name, room),
        adjacent_solids(Kitchen, Dining).

rhs(qa_11, [Dining]):-
        set_state(stair),
        kill_label(Dining, name, room),
        make_label(Dining, name, dining),
                dining_room_color(DColor),
                set_solid_color(Dining, DColor).
```

Rule qa_16 creates a 2nd floor room on top of each 1st floor room. It looks for any first floor room except the stairway, and stacks a room on top of the first floor room. Rule qa_16 is illustrated with Figure 8.27.

```
description(qa_16, 'Add a second floor room above a existing room.').

lhs(qa_16, [Top, Height, Room], [Room]):-
        state(second),
        label(Room, name, S),
        member(S, [room, parlor, kitchen, dining, hall, pantry]),
        not(label(Room, mark, stacked)),
        top(Top, Room),
        room_height(Height).

rhs(qa_16, [Top, Height, Room]):-
        stack_solid(Top, Height, NewRoom),
        make_label(Room, below, NewRoom),
        make_label(NewRoom, floor, second),
        room_color(RColor),
        set_solid_color(NewRoom, RColor).
```

The Queen Anne grammar constructs a geometric representation as well as symbolic information about the geometric entities. For example, some solid represent rooms and are intended for a specific purpose (kitchen). This information is recorded in the label database.

In order to model the current state of many Queen Anne houses, additional rules are needed to convert these single family residences into apartments for subsequent rental to graduate students.

Figure 8.14: Two mountains.

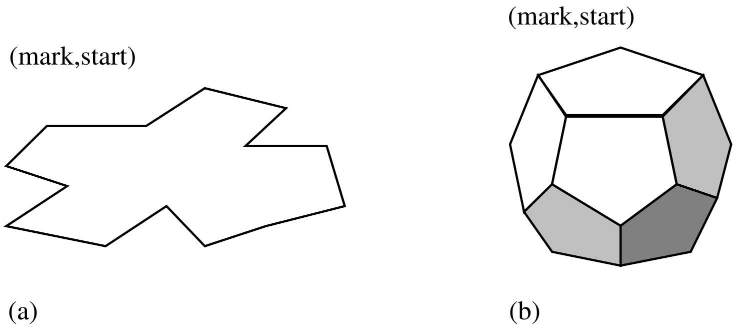


Figure 8.15: Possible initial solids for spiral grammars.

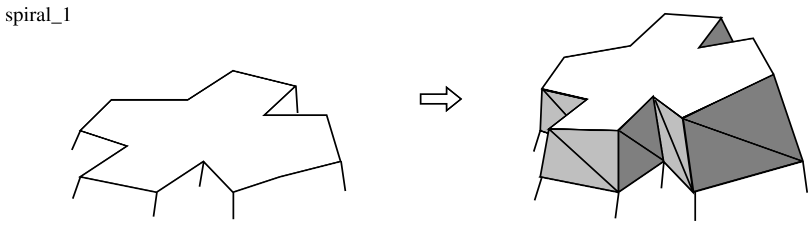


Figure 8.16: Rule for the spiral grammars.

Figure 8.17: A spiral form.

Figure 8.18: An icosopus egg.

Figure 8.19: A developing spiny icosopus (without spines).

Figure 8.20: A developing spiny icosopus (with spines).

Figure 8.21: An adult spiny icosopus.

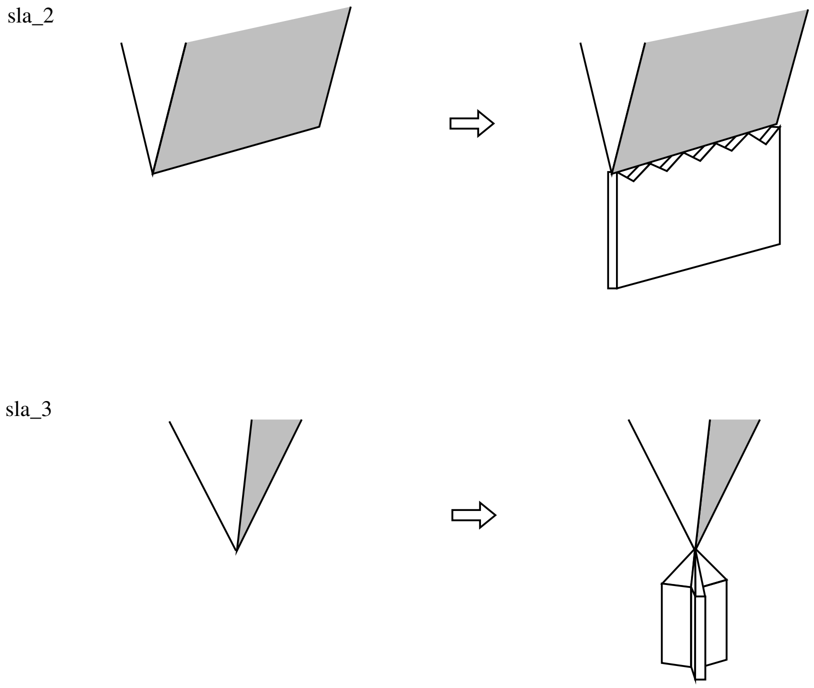


Figure 8.22: Rules for locating vertex and edge supports.

Figure 8.23: Stereo lithography supports for a solid.

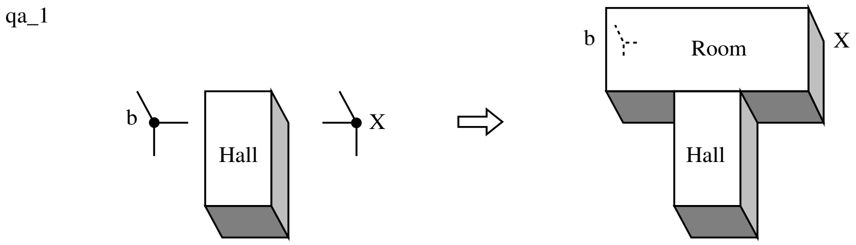


Figure 8.24: A basic layout rule for Queen Anne houses.

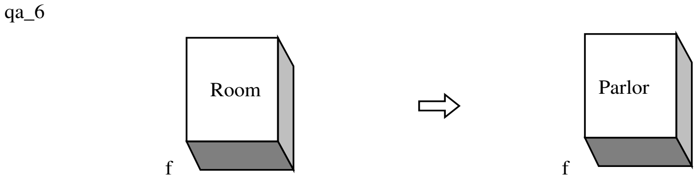


Figure 8.25: A rule for locating the parlor in a Queen Anne house.

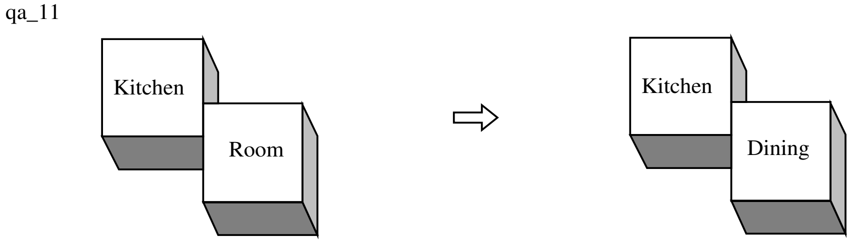


Figure 8.26: A rule for locating the dining room in a Queen Anne house.

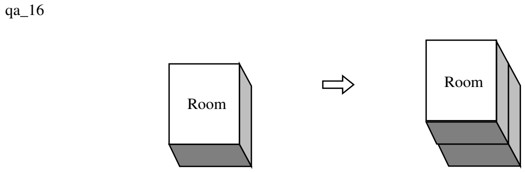


Figure 8.27: A rule for adding the second floor to a Queen Anne house.

Figure 8.28: Two Queen Anne houses.

Figure 8.29: Additional Queen Anne houses.

Chapter 9

Genesis : A Boundary Solid Grammar Interpreter

Genesis is an implementation of the boundary solid grammar formalism. It provides facilities for representation and display of solids, match conditions, solid modeling operations, rule and grammar definition, and searching through the language of a grammar. **Genesis** has been under development since December 1988. The first version was operational in early January 1989, and three distinct versions have preceded the current implementation.

This chapter introduces the **Genesis** boundary solid grammar interpreter, including its user interface and modeling facilities, system architecture, and performance. A detailed discussion of **Genesis** can be found in the **Genesis** Reference Manual [20].

9.1 Using Genesis

Users interact with **Genesis** in several ways. They may use **Genesis** as a boundary representation solid modeler, using local operations, unary shape operations, and Boolean operations. **Genesis** provides facilities for defining matching conditions and operations, defining rules, constructing initial solids, applying rules, and searching for derivations in the language of a grammar. These facilities allow the user to experiment with generative grammars, creating alternative initial solids, defining new rules, and applying rules in different ways to generate interesting and useful models.

Users can interactively locate features of solids, and modify solids with operations. They can construct conditions and operations as described in Chapter 7. A user can then use these conditions and operations to define solid rules. Solid rules are described with three components: a description, a left-hand side, and a right-hand side. The description is a textual description of the rule that will be presented to the user as the rule is matched. The left-hand side is the set of match conditions that must be satisfied in order to apply the rule. The right-hand side is the sequence of operations and match conditions that transform the matched solids.

There are several ways to generate solids using the rules that users have defined. The primary facilities are describe below.

- Apply a specific rule once and display the results. This is useful primarily for debugging new rules.
- Apply the applicable rules to the current solids. **Genesis** displays the current solids with the matched elements highlighted for a given match, and the user is queried to allow or disallow the application of the rule.
- Apply any applicable rules a given number of times (depth-first) and display the result after each rule application.

Genesis provides several ways to search through the space of derivations of a grammar.

- Search depth-first for a member of the language of the grammar (which occurs when the current state equal to `done`).
- Search depth-first for a derivation of the grammar with has a given state `State` as the current state.
- Search randomly for a member of the language of the grammar (with the current state equal to `done`). A given rule is applied if a random number is (between 0 and 1) is less than `Probability`.
- Search randomly for a derivation of a grammar which has a given state `State` as the current state.

A user may mix these various interaction modes. For example, one may apply rules to generate a model, then manually modify the model using solid modeling operations. The new model may then be used as an initial solid for further detailing and development with another rule set. A user may also generate a model with another solid modeler, read it into **Genesis** from a file, and use it as an initial solid in a grammar to generate additional detail, or related solids.

9.2 Genesis System Architecture

Genesis consists of a boundary representation solid modeler, a database for non-geometric information, a logic programming interpreter/complier, graphics facilities for display of solids, and a graphical user interface. **Genesis** was written by the author in the C programming language, and consists of approximately 20,000 lines of C code, including the solid modeler, graphics, and

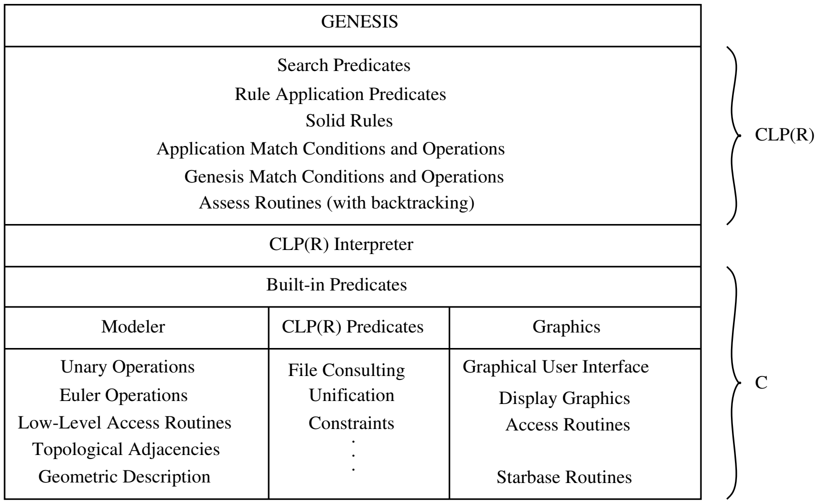


Figure 9.1: The architecture of **Genesis** .

label database. Portions of the rule interpreter are constructed on top of **CLP**($\mathcal{R}$) , implemented with about 7,000 lines of **CLP**($\mathcal{R}$)source code. (The IBM **CLP**($\mathcal{R}$) Complier consists of an additional 15,000 lines of C.) The architecture of the current implementation of **Genesis** is presented in Figure 9.1. The data structures used within **Genesis** are illustrated in Figure 9.2.

Solid Modeler

The solids modeler in **Genesis** uses the generalized split-edge data structure to represent manifold and nonmanifold solids (described in Chapter 2). Routines provide access from any element to any of its adjacent elements, with most of the accesses in constant time. A complete set of manifold and nonmanifold Euler operations are provided, including the nonmanifold Euler operators (described in Chapter 4). The unary shape operations are provided, and Boolean operations are computed using the unary shape operations.

Label Database

The labeling mechanism of the boundary solid grammar formalism allows non-geometric information to be associated with topological elements. A database for this non-geometric information is provided in **Genesis** . In order to provide efficient access, the database has indexes directly from topological elements into the label database.

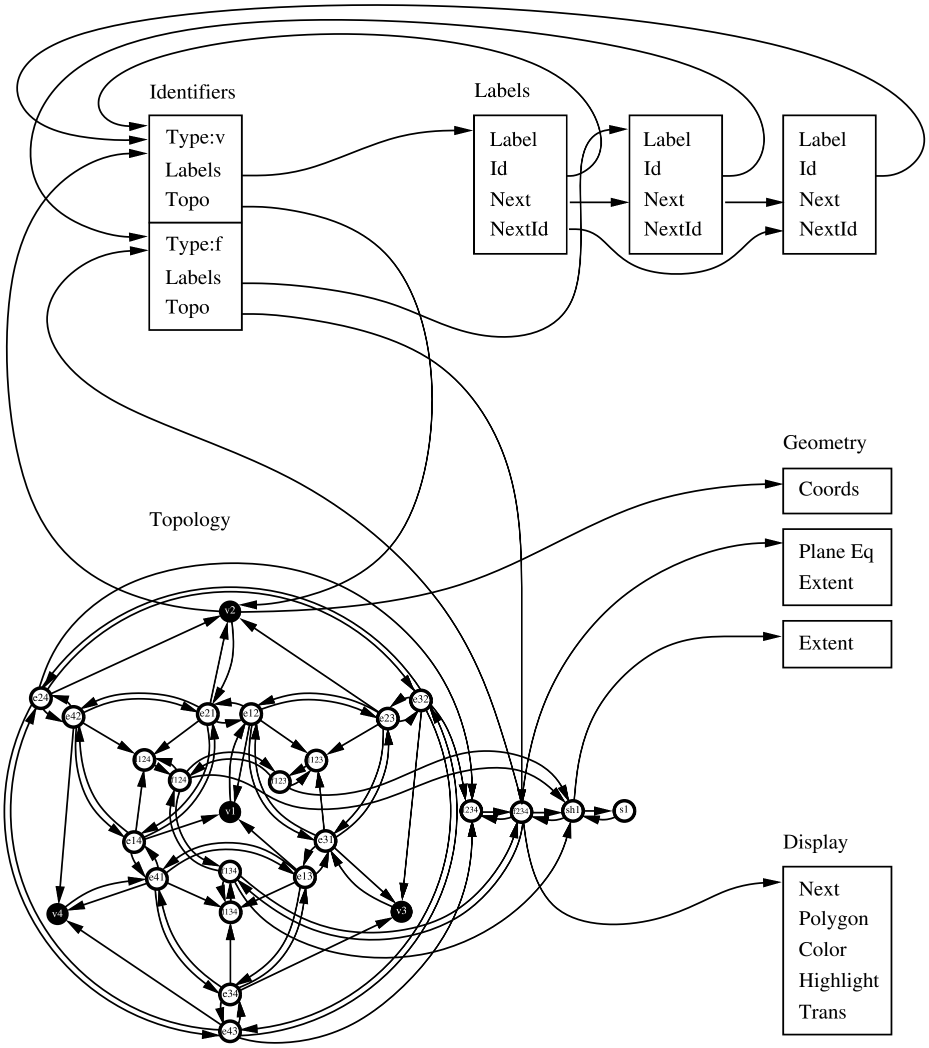


Figure 9.2: Data structures within **Genesis** (a simplified view).

Graphics and User Interface

A graphical user interface allows the user to move the position and direction of the camera and viewpoint, modify the brightness, color, and position of directional lights, modify the brightness and color of the ambient light, and modify the amount of specular reflection and color of specular highlights. The user interface also controls the culling of back-facing polygons, depth-cueing, the display of objects with wireframe or shaded surface rendering, the default color of objects, and the background color. The graphical user interface is constructed with widgets in the X window system.

The graphics routines are written in C. These routines access and maintain additional structures for polygon display. Hidden surface removal is accomplished using a hardware z-buffer, and HP Starbase graphics routines.

Logic Programming Interpreter / Complier

Genesis uses IBM's compiler-based implementation of the **CLP**($\mathcal{R}$) programming language [24]. Rule matching and application is accomplished using three mechanisms. At the lowest level, primitive access of the boundary representation, primitive match conditions, and modeling operation are accomplished in the **Genesis** solid modeler. These routines are accessed by the **CLP**($\mathcal{R}$) complier as built-in predicates. Access to the label database is also accomplished this way. The graphics routines are called as built-in predicates, and interact directly with the solid modeler. At the next higher level, unification and backtracking mechanisms are provided within the **CLP**($\mathcal{R}$) complier. **CLP**($\mathcal{R}$) code provides the access and backtracking of the low level modeling representation and operations. Complex matching conditions and operations, rule application, and search facilities are constructed on this framework (written in **CLP**($\mathcal{R}$)).

9.3 Performance

Grammar	Rule Applications	Faces	Minutes
Uniform Snowflake	500	2504	3:26
Mountain	25	87	0:09
Mountain	100	312	0:29
Mountain	300	912	1:35
Mountain	500	1512	3:19
Mountain	1000	3012	13:47
Mountain	3500	10512	104:20
Mountain	8000	24012	352:00
Queen Anne	67	1218	2:35
Queen Anne (no graphics)	67	1218	2:03

Table 9.1: Performance benchmarks on **Genesis** .

Genesis constructs a detailed Queen Anne house, composed of about 135 solids and 1200 faces, in 2 to 3 minutes (clock time). This time includes updating the graphical display of the model after each of approximately 70 rule applications. Generating a mountain with 1500 faces takes about 3 minutes. Mountain models with over 24000 faces have been generated with **Genesis** (see Figure 8.14). Table 9.1 summarized the times needed to generate derivations of various grammars. These performance figures are for **Genesis** running on a Hewlett-Packard 835 workstation configured with 32 Megabytes of physical memory and 160 Megabytes of virtual memory. They include graphical display of the solids after each rule application, with hidden surface removal and three directional light sources plus ambient light, except as noted.

Part III

Conclusion

Chapter 10

Contributions of the Thesis

This research was undertaken to investigate one of "the grand challenges in three-dimensional graphics ... to make simple modeling easy and to make complex modeling accessible to far more people" [49]. In this light, the thesis has provided a detailed look at the fundamental concepts and operations of solid modeling, and the automatic generation of solid models based on a grammatical paradigm. This chapter considers the material of the thesis and discusses the various contributions to theory, implementations, and applications.

10.1 Formalizations

The thesis makes several contributions to a formal understanding of topics in areas of solid modeling and generative geometric design. These contributions are listed below:

- A formal definition of boundary solid grammars
- The unary shape operations
- The generalized split-edge representation
- Generalized Euler operations
- A generalized form of the Euler-Poincaré formula

A formal definition of boundary solid grammars was presented. This includes a boundary representation for solids, a paradigm for reasoning about solids, a method of constructing operators, definitions of solid rules and boundary solid grammars, and a formal method for generating languages of solids.

The unary shape operations allow any topologically valid boundary model to be interpreted as a valid boundary representation. The regularized Boolean operations may be computed using the unary shape operations. The use of local operations, the unary shape operations, and Boolean operations on boundary representations constitutes a new solid modeling approach.

A new boundary representation has been introduced for the representation of (regular) nonmanifold solids, the generalized split-edge representation.

Generalized Euler operations have been presented to allow direct manipulation of the nonmanifold topological adjacencies of boundary representations of solids.

A generalized form of the Euler-Poincaré formula has been introduced, expressing the relationship between the boundary elements of manifold and nonmanifold solids.

10.2 Implementations

The research undertaken for this thesis has produced some practical results. These contributions take the form of usable software and algorithms, and are listed below:

- **Genesis** boundary solid grammar interpreter
- **Genesis** solid modeler
- Unary shape operations algorithm

The thesis presents **Genesis** , an implementation of a boundary solid grammar interpreter.

A solid modeler representing manifold and nonmanifold solids has been presented. The modeler provides local modification of the topology using both traditional (manifold) and nonmanifold Euler operations.

An algorithm for the computation of the unary shape operations has been presented in the thesis. This algorithm has been implemented in the **Genesis** boundary solid grammar interpreter.

10.3 Applications

The thesis presents a number of grammars to demonstrate the concepts and usefulness of the formalism.

- Various geometric forms
- Stereo lithography support structures

- Queen Anne houses

Grammars generate simple geometric forms including snowflakes, recursive octahedra, mountains, and spirals. Automated construction of supports for solid models being built using stereo lithography is accomplished using another boundary solid grammar. An extensive grammar is presented that characterizes Queen Anne houses.

Chapter 11

Future Directions

The work presented here has uncovered at least as many interesting topics as it has addressed. In this chapter, I present some of these topics and comment on their nature.

11.1 Generative Grammars

There are many interesting and useful areas for investigation in the area of generative grammars. These generally fall into two areas: alternative representations, and user interfaces.

- Boundary solid grammars using non-linear representations
- Boundary solid grammars using non-homogeneous representations
- Boundary solid grammars using constrained models
- Alternative search strategies

Boundary solid grammars that use curved surface representations would provide capabilities needed in a large variety of engineering and manufacturing domains. There is enormous potential for automating the design of manufacturing dies and equipment from the engineered part descriptions. Additional work is needed to allow parametric matching on curve surfaces, and provide facilities for modifying the surfaces.

There has been much recent interest in non-homogeneous ("nonmanifold") representations. Non-homogeneous modelers allow solid, surface, and wireframe models to be incorporated into a single representation. Initial investigations indicate that the boundary solid grammar formalism is amenable to non-homogeneous representations. Grammars could then be used, for example, to generate 3-D models from a 1-D or 2-D abstraction.

It would be interesting to develop grammar systems that allow the representation and generation of constrained models without grounded geometry. The user of such a system would then explore both the space of constrained models and the limits of the constraints of individual models.

Generating models with constrained geometry will require addressing difficult problems. First of all, the number of possible rule matches explodes as models become less constrained. There is a combinatorial explosion of the size of the space of designs as we allow additional degrees of freedom in the design. This seems to be an inherent problem in design. New tools will be needed to allow a designer to selectively constrain and explore the space of possible designs generated by a grammar of this type. Another difficulty is in displaying the constrained models. It is clear how to display a model with grounded geometry. However, when our model is not fully constrained, then it is difficult to convey to the user of the system what range of instances is described.

To allow a user to better explore the language of a grammar, alternative search strategies would be useful. First, a breadth-first strategy would allow a user to examine all the derivations generated by a given rule. This would be especially useful for testing rules. Breadth-first search would also be useful for viewing the entire language of a grammar (when sufficiently small). For designing, using and branch and bound or best-first strategy would allow a more focused search. These later strategies would require the definition of some (domain dependent) evaluation criteria. The most useful approach will probably require several of these search techniques to be used simultaneously.

11.2 Solid Modeling

The most immediate directions leading from the thesis involve the implementation and use of the unary shape operations.

- Unary shape operations
- Incremental unary shape operations
- Unary shape operations on non-linear boundary representations
- Unary shape operations on non-homogeneous representations
- Offsetting operations

Additional work is needed to show the soundness of the unary shape operations. The proofs presented in the thesis are intended to provide outlines of the necessary arguments, but do not constitute rigorous proofs. The thesis reports an algorithm for the computation of the generalized unary intersection. Additional algorithm development is desirable, as well as presentation of proofs of correctness and the computational complexity of the algorithms.

An incremental form of the unary shape operations could determine which elements of a solid could have intersections due to local modifications of geometry and topology that have occurred since the most recent unary shape operation. This could be accomplished by recording the local modifications that have been made since the most recent unary shape operation and recording which element could be affected. Since the operation would not need to consider the intersection of every element with every other, this would reduce the total number of comparisons and thereby reduce the overall complexity.

Algorithms are needed for computing the generalized unary intersection for non-linear surfaces. In particular, the generalized unary intersection will be very useful when applied to NURBS (non-uniform rational b-spline) surfaces. This would provide the foundation for integrating geometric (surface) modeling with boundary representation solid modeling.

It would be useful to incorporate the unary shape operations into non-homogeneous representations. Unary shape operations would allow local modifications of elements of non-homogeneous models, while guaranteeing that the resulting models can be returned to a valid state. This would require a non-regularized definition of the unary shape operations, including a characterization of the winding number (or something like it) for points on the bounding surfaces.

When a sphere is moved across a surface, it produces a volume bounded by the offset of the surface. Offsetting can be viewed as an operations on solids or on surfaces, and has been used to define global *blending* operations on solids in which all edges are rounded or filleted.

Much of the difficulty of constructing offsetting operations is caused by the potential of self-intersections of the offset surfaces. Instead of viewing offsetting operations as global operation, it may be useful to consider offsetting as two separate operations: local offsetting of the surface of a given solid, followed by the unary union operation applied to the offset surface. The local offsetting would compute the offset surface, with possible self-intersections. The unary union would take the offset surface and resolve the self-intersections to produce the offset solid.

Appendix A

Euler Operators as Graph Productions

This appendix presents graph productions of Euler operations on our b-graph representation of solids. The Euler operations included here create minimal solids (`msflv`), insert strut edges (`mev`), split edges (`esplit`), and split faces (`mefl`). A detailed discussion of the Euler operations may be found in Chapter 4.

A graph production has three parts: a subgraph that will be matched and removed from the graph being matched; a subgraph that will be inserted in place of the removed subgraph; and an embedding which specifies the connections between the inserted subgraph and the original graph. Although the subgraph matching, removal, and substitution are straight-forward, the embedding requires further explanation. The embedding is specified using graph operators, and these graph operators are defined below. The notation and definitions for graph operators and graph production are adapted from Levy and Yueh [30], which was originally introduced by Nagl [36, 37].

Definition A.1 *For the node set K of any labeled boundary graph with the node set K' of a given subgraph and any node $x \in K$, the* **graph operator** *A yields a subset of K, written as $A(x)$ according to the following recursive definition:*

1. *$L_i(x) = \{y \in K - K' \mid (y, x) \in \rho_i\}$ specifies the set of nodes with i-labeled arcs that terminate in x.*

2. *$R_i(x) = \{y \in K - K' \mid (x, y) \in \rho_i\}$ denotes the set of nodes with i-labeled arcs leaving x.*

3. *$AB(x) = \{y \mid y \in K - K', \exists z(z \in K - K', z \in B(x), y \in A(z))\}$ specifies the subset of nodes of $K - K'$ which are generated by sequencing graph operators.*

4. $(A \cup B)(x) = A(x) \cup B(x)$,
specifies the subset of nodes of $K - K'$ which are generated by the branching of operators.

Definition A.2 *A* **graph production** *is a triple $P = (b_l, b_r, E)$ with $b_l, b_r \in b(\Sigma_\star)$ and an embedding transformation $E = ((l_a, r_a))_{a \in \Sigma_{arc}}$, with the embedding components l_a and r_a having the following form:*

$$l_a = \bigcup_{j=1}^{p} A_j(y_j) \times \{z_j\}$$

$$r_a = \bigcup_{j=1}^{q} \{z_j\} \times A_j(y_j)$$

where $y_j \in K_l$, $z_j \in K_r$, A_j is an graph operator, and $p, q \geq 1$.

The labeled boundary graph b_l is the left hand side and b_r the right hand side of the graph production. K_l, K_r are the sets of nodes in b_l, b_r, respectively.

We may abbreviate $A(y) \times \{z\}$ by $(A(y); z)$ and $\{z\} \times A(y)$ by $(z; A(y))$. $(A; a, b)$ is used to represent $(A; a) \cup (A; b)$ with a similar notation for r-operators.

Informally, l_a specifies the a-labeled arcs from the existing graph to the newly inserted subgraph. $A_j(y_j)$ selects a subset of nodes in $K - K_l$, and connects these nodes to each of the nodes $\{z_j\}$ in the new subgraph with an a-labeled arc. Similarly, r_a specifies the a-labeled arcs from the subgraph to the existing graph.

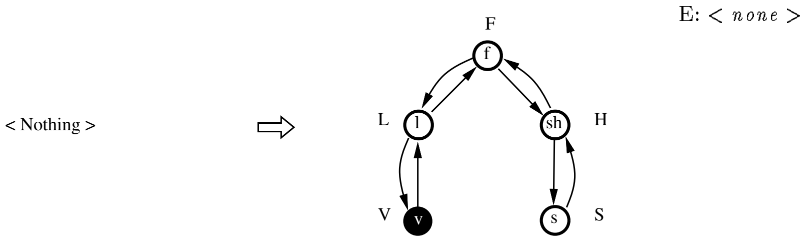
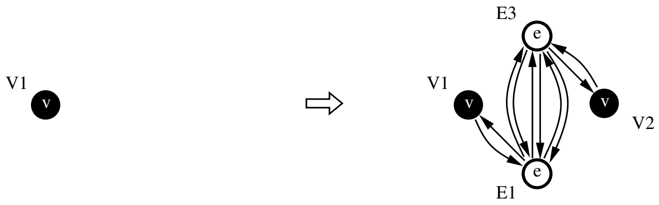


Figure A.1: The `msflv` graph production.

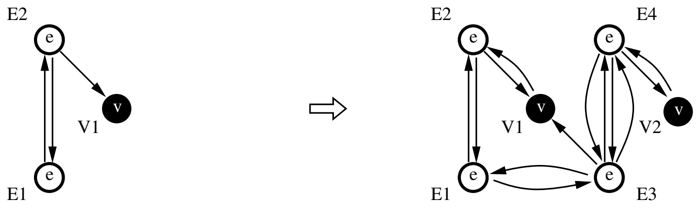
Graph productions implementing Euler operators for the generalized split-edge data structure are presented in Figures A.1 through A.6. These graph productions are used as operators by requiring that the nodes of b_l are bound to specific nodes in a b-graph, as specified by the operators.

Definition A.3 *A labeled boundary graph $b' \in b(\Sigma_\star)$ is* **directly derivable** *from another $b \in b(\Sigma_\star)$ by an operator O (abbreviated $b \underset{O}{\Longrightarrow} b'$) iff*



$$\begin{aligned} \text{E:}\quad & r_{edgeh\_l} && = (\mathrm{E}_1, \mathrm{E}_3; \textsc{loop}L_{loop\_v}(\mathrm{V}_1)) \\ & l_{loop\_eh} && = (\textsc{loop}L_{loop\_v}(\mathrm{V}_1); \mathrm{E}_1) \end{aligned}$$

Figure A.2: The `mev` graph production when there are no edges in the loop.



$$\begin{aligned} \text{E:}\quad & r_{edgeh\_v} && = (\mathrm{E}_1; \textsc{vertex}R_{edgeh\_v}(\mathrm{E}_1)) \\ & l_{cw\_eh} && = (\textsc{edgehalf}L_{cw\_eh}(\mathrm{E}_1); \mathrm{E}_1) \\ & l_{cw\_eh} && = (\textsc{edgehalf}L_{cw\_eh}(\mathrm{E}_2); \mathrm{E}_2) \\ & r_{cw\_eh} && = (\mathrm{E}_4; \textsc{edgehalf}R_{cw\_eh}(\mathrm{E}_1)) \\ & r_{cw\_eh} && = (\mathrm{E}_2; \textsc{edgehalf}R_{cw\_eh}(\mathrm{E}_2)) \\ & l_{ccw\_eh} && = (\textsc{edgehalf}L_{ccw\_eh}(\mathrm{E}_1); \mathrm{E}_4) \\ & l_{ccw\_eh} && = (\textsc{edgehalf}L_{ccw\_eh}(\mathrm{E}_2); \mathrm{E}_2) \\ & r_{ccw\_eh} && = (\mathrm{E}_1; \textsc{edgehalf}R_{ccw\_eh}(\mathrm{E}_1)) \\ & r_{ccw\_eh} && = (\mathrm{E}_2; \textsc{edgehalf}R_{ccw\_eh}(\mathrm{E}_2)) \\ & r_{edgeh\_l} && = (\mathrm{E}_1, \mathrm{E}_3, \mathrm{E}_4; \textsc{loop}R_{edgeh\_l}(\mathrm{E}_1)) \\ & r_{edgeh\_l} && = (\mathrm{E}_2; \textsc{loop}R_{edgeh\_l}(\mathrm{E}_2)) \\ & l_{loop\_eh} && = (\textsc{loop}L_{loop\_eh}(\mathrm{E}_1); \mathrm{E}_1) \\ & l_{loop\_eh} && = (\textsc{loop}L_{loop\_eh}(\mathrm{E}_2); \mathrm{E}_2) \end{aligned}$$

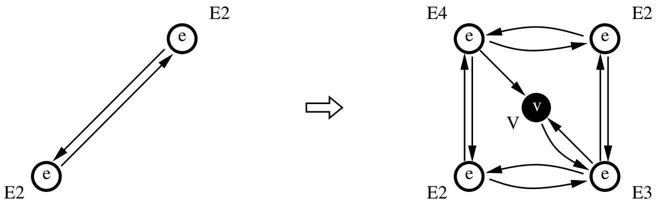
Figure A.3: The general `mev` graph production.

1. $b_l \subseteq b$ *and* $b_r \subseteq b'$*;*

2. $b - b_l = b' - b_r$*; and*

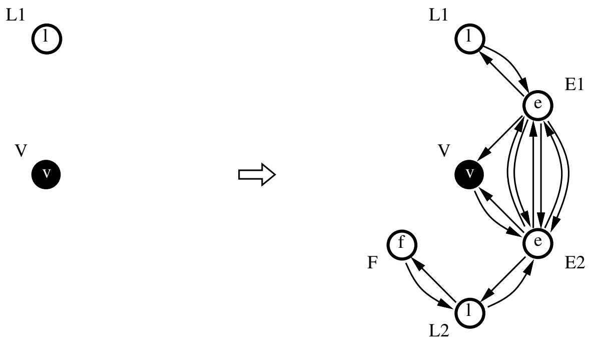
3. $In_a(b_r, b') = l_a$ *and* $Out_a(b_r, b') = r_a$ *for* $a \in \Sigma_{arc}$

where $In_a(b_r, b')$ *is the set of all a-labeled incoming arcs originating in* $b' - b_r$ *and terminating in* b*, and* $Out_a(b_r, b')$ *is the set of all a-labeled outgoing arcs originating in* b_r *and terminating in* $b - b_r$*.*



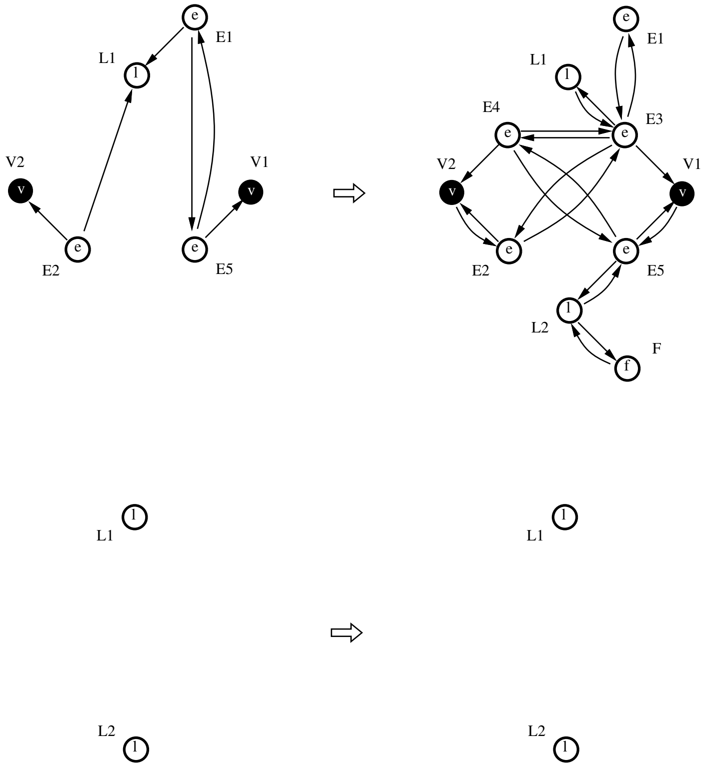
$$\begin{aligned}
\text{E:}\quad & r_{edgeh\_v} && = (\mathrm{E}_1; \textsc{vertex} R_{edgeh\_v}(\mathrm{E}_1)) \\
& r_{edgeh\_v} && = (\mathrm{E}_2; \textsc{vertex} R_{edgeh\_v}(\mathrm{E}_2)) \\
& l_{cw\_eh} && = (\textsc{edgehalf} L_{cw\_eh}(\mathrm{E}_1); \mathrm{E}_1) \\
& l_{cw\_eh} && = (\textsc{edgehalf} L_{cw\_eh}(\mathrm{E}_2); \mathrm{E}_2) \\
& r_{cw\_eh} && = (\mathrm{E}_4; \textsc{edgehalf} R_{cw\_eh}(\mathrm{E}_2)) \\
& r_{cw\_eh} && = (\mathrm{E}_3; \textsc{edgehalf} R_{cw\_eh}(\mathrm{E}_1)) \\
& l_{ccw\_eh} && = (\textsc{edgehalf} L_{ccw\_eh}(\mathrm{E}_1); \mathrm{E}_3) \\
& l_{ccw\_eh} && = (\textsc{edgehalf} L_{ccw\_eh}(\mathrm{E}_2); \mathrm{E}_4) \\
& r_{ccw\_eh} && = (\mathrm{E}_1; \textsc{edgehalf} R_{ccw\_eh}(\mathrm{E}_1)) \\
& r_{ccw\_eh} && = (\mathrm{E}_2; \textsc{edgehalf} R_{ccw\_eh}(\mathrm{E}_2)) \\
& r_{edgeh\_l} && = (\mathrm{E}_1, \mathrm{E}_3; \textsc{loop} R_{edgeh\_l}(\mathrm{E}_1)) \\
& r_{edgeh\_l} && = (\mathrm{E}_2, \mathrm{E}_4; \textsc{loop} R_{edgeh\_l}(\mathrm{E}_2)) \\
& l_{loop\_eh} && = (\textsc{loop} L_{loop\_eh}(\mathrm{E}_1); \mathrm{E}_1) \\
& l_{loop\_eh} && = (\textsc{loop} L_{loop\_eh}(\mathrm{E}_2); \mathrm{E}_2)
\end{aligned}$$

Figure A.4: The `esplit` graph production.



$$\begin{aligned}
\text{E:}\quad & l_{face\_l} && = (\textsc{face} L_{face\_l}(\mathrm{L}_1); \mathrm{L}_1) \\
& r_{loop\_f} && = (\mathrm{L}_1; \textsc{face} R_{loop\_f}(\mathrm{L}_1)) \\
& r_{face\_sh} && = (\mathrm{F}_1; \textsc{shell} R_{face\_sh} R_{loop\_f}(\mathrm{L}_1))
\end{aligned}$$

Figure A.5: The `mefl` graph production when there are no edges in the loop.



$$\begin{array}{lll}
\text{E:} & r_{edgeh\_v} & = (\mathrm{E}_1;\textsc{vertex}R_{edgeh\_v}(\mathrm{E}_1)) \\
 & l_{edgeh\_v} & = (\textsc{edgehalf}L_{edgeh\_v}(\mathrm{V}_1);\mathrm{V}_1) \\
 & l_{edgeh\_v} & = (\textsc{edgehalf}L_{edgeh\_v}(\mathrm{V}_2);\mathrm{V}_2) \\
 & l_{cw\_eh} & = (\textsc{edgehalf}L_{cw\_eh}(\mathrm{E}_1);\mathrm{E}_1) \\
 & l_{cw\_eh} & = (\textsc{edgehalf}L_{cw\_eh}(\mathrm{E}_2);\mathrm{E}_4) \\
 & r_{cw\_eh} & = (\mathrm{E}_2;\textsc{edgehalf}R_{cw\_eh}(\mathrm{E}_2)) \\
 & r_{cw\_eh} & = (\mathrm{E}_5;\textsc{edgehalf}R_{cw\_eh}(\mathrm{E}_5)) \\
 & l_{ccw\_eh} & = (\textsc{edgehalf}L_{ccw\_eh}(\mathrm{E}_5);\mathrm{E}_5) \\
 & l_{ccw\_eh} & = (\textsc{edgehalf}L_{ccw\_eh}(\mathrm{E}_2);\mathrm{E}_2) \\
 & r_{ccw\_eh} & = (\mathrm{E}_1;\textsc{edgehalf}R_{ccw\_eh}(\mathrm{E}_1)) \\
 & r_{ccw\_eh} & = (\mathrm{E}_4;\textsc{edgehalf}R_{ccw\_eh}(\mathrm{E}_2)) \\
 & l_{other\_eh} & = (\textsc{edgehalf}L_{other\_eh}(\mathrm{E}_1);\mathrm{E}_1) \\
 & l_{other\_eh} & = (\textsc{edgehalf}L_{other\_eh}(\mathrm{E}_2);\mathrm{E}_2) \\
 & l_{other\_eh} & = (\textsc{edgehalf}L_{other\_eh}(\mathrm{E}_5);\mathrm{E}_5) \\
 & r_{other\_eh} & = (\mathrm{E}_1;\textsc{edgehalf}R_{other\_eh}(\mathrm{E}_1)) \\
 & r_{other\_eh} & = (\mathrm{E}_2;\textsc{edgehalf}R_{other\_eh}(\mathrm{E}_2)) \\
 & r_{other\_eh} & = (\mathrm{E}_5;\textsc{edgehalf}R_{other\_eh}(\mathrm{E}_5)) \\
 & l_{face\_l} & = (\textsc{face}L_{face\_l}(\mathrm{L}_1);\mathrm{L}_1) \\
 & r_{loop\_f} & = (\mathrm{L}_1;\textsc{face}R_{loop\_f}(\mathrm{L}_1)) \\
 & r_{face\_sh} & = (\mathrm{F};\textsc{shell}R_{face\_sh}R_{loop\_f}(\mathrm{L}_1))
\end{array}$$

$$\begin{array}{lll}
\text{E:} & l_{edgeh\_l} & = (\textsc{edgehalf} \\
 & & \quad \cup_{i=1}^{n} (L_{cw\_eh}L_{edgeh\_l}(\mathrm{L}_1);\mathrm{L}_1) \\
 & l_{edgeh\_l} & = (\textsc{edgehalf} \\
 & & \quad \cup_{i=1}^{n} (L_{cw\_eh}L_{edgeh\_l}(\mathrm{L}_2);\mathrm{L}_2) \\
 & r_{loop\_eh} & = (\mathrm{L}_1;\textsc{edgehalf}R_{loop\_eh}(\mathrm{L}_1)) \\
 & r_{loop\_eh} & = (\mathrm{L}_2;\textsc{edgehalf}R_{loop\_eh}(\mathrm{L}_2)) \\
 & l_{face\_l} & = (\textsc{face}L_{face\_l}(\mathrm{L}_1);\mathrm{L}_1) \\
 & l_{face\_l} & = (\textsc{face}L_{face\_l}(\mathrm{L}_2);\mathrm{L}_2) \\
 & r_{loop\_f} & = (\mathrm{L}_1;\textsc{face}R_{loop\_f}(\mathrm{L}_1)) \\
 & r_{loop\_f} & = (\mathrm{L}_2;\textsc{face}R_{loop\_f}(\mathrm{L}_2))
\end{array}$$

Figure A.6: General `mefl` as two sequentially applied graph productions.

Bibliography

[1] B. G. Baumgart. Winged-edge polyhedron representation. Technical Report CS-320, Stanford AI Laboratory, Stanford University, October 1972.

[2] B. G. Baumgart. A polyhedron representation for computer vision. In *AFIPS Conf. Proc.*, volume 44, pages 589–596, 1975.

[3] I. C. Braid. *Designing with Volumes.* PhD thesis, Computer-Aided Design Group, University of Cambridge, Cambridge, England, February 1973.

[4] I. C. Braid, R. C. Hillyard, and I. A. Stroud. Stepwise construction of polyhedra in geometric modeling. In K. W. Brodlie, editor, *Mathematical Methods in Computer Graphics and Design*, pages 123,141. Academic Press, New York, 1980.

[5] Horst Bunke. Programmed graph grammars. In *Lecture Notes in Computer Science*, volume 56, pages 155–166, Berlin, 1977. Springer-Verlag.

[6] Horst Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(6):574–582, 1982.

[7] W. G. Chinn and N. E. Steenrod. *First Concepts of Topology.* Random House, New York, 1966.

[8] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1984.

[9] Leila De Floriani. Feature extraction from boundary models of three-dimensional objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(8):785–798, 1989.

[10] Philippe de Reffye, Claude Edelin, Jean Francon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. *Computer Graphics*, 22(4):151–158, 1988.

[11] H. Desaulniers and N. F. Stewart. Generalized Euler operators for r-sets. Technical Report 649, Department d'Informatique et de Recherche Operationnelle, Universite de Montreal, April 1988.

[12] F. Downing and U. Flemming. The bungalows of Buffalo. *Environment and Planning B*, 8:269–293, 1981.

[13] C. Eastman, J. Lividini, and D. Stoker. A database for designing large physical systems. In *Proc. 1975 National Computer Conference*, pages 603–611. AFIPS Press, New Jersey, 1975.

[14] C. M. Eastman and K. Weiler. Geometric modeling using the euler operators. In *Proc. First Ann. Conf. Computer Graphics and CAD/CAM Systems*, pages 248–254, Cambridge, Mass., April 1979. M.I.T. Press.

[15] Patrick Fitzhorn. A linguistic formalism for engineering solid modeling. In *Graph-Grammars and Their Application to Computer Science*, pages 202–215, Berlin, 1987. Springer-Verlag.

[16] U. Flemming. The secret of the Casa Guiliani Frigerio. *Environment and Planning B*, 8:87–96, 1981.

[17] U. Flemming. More than the sum of parts: the grammar of Queen Anne houses. *Environment and Planning B*, 14:323–350, 1987.

[18] U. Flemming, Robert Coyne, Shakunthala Pithavadian, and Raymond Gindroz. A pattern book for Shadyside. Technical report, Department of Architecture, Carnegie Mellon University, December 1985.

[19] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[20] Jeff A. Heisserman. Genesis reference manual. Technical Report EDRC 48-??-91, Engineering Design Research Center, Carnegie Mellon University, August 1991.

[21] Michael Henle. *A Combinatorial Introduction to Topology.* W. H. Freeman and Company, San Francisco, 1979.

[22] Robin Hillyard. The build group of solid modelers. *IEEE Computer Graphics and Applications*, 2(2):43–52, March 1982.

[23] Christoph M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann Publishers, San Mateo, 1989.

[24] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Rolland H. C. Yap. The CLP($\Re$) language and system. Technical Report CMU-CS-90-181, Department of Computer Science, Carnegie Mellon University, October 1990.

[25] Michael Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, Department of Computer Science, McGill University, Montreal, Canada, November 1988.

[26] T. W. Knight. The generation of Hepplewhite-style chair-back designs. *Environment and Planning B*, 7:227–238, 1980.

[27] T. W. Knight. The forty-one steps. *Environment and Planning B*, 8:97–114, 1981.

[28] H. Koning and J. Eizenberg. The language of the prairie: Frank Lloyd Wright's prairie houses. *Environment and Planning B*, 8:295–323, 1981.

[29] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes. Constructive solid geometry for polyhedral objects. *Computer Graphics*, 20(4):161–170, 1986.

[30] L. S. Levy and Kang Yueh. On labelled graph grammars. *Computing*, 20:109–125, 1978.

[31] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[32] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman, New York, 1983.

[33] M. Mantyla and R. Sulonen. GWB: A solid modeler with Euler operators. *IEEE Computer Graphics and Applications*, 2(7):17–31, September 1982.

[34] Martii Mantyla. A note on the modeling space of Euler operators. *Computer Vision, Graphics, and Image Processing*, 26(1):45–60, April 1984.

[35] Martti Mantyla. Boolean operations of 2-manifolds through vertex neighborhood classification. *ACM Transactions on Graphics*, 5(1):1–29, January 1986.

[36] M. Nagl. Formal languages of labelled graphs. *Computing*, 16:113–137, 1976.

[37] M. Nagl. Graph rewriting systems and their application in biology. In *Lecture Notes in Biomathematics*, volume 11, pages 135–156, Berlin, 1976. Springer-Verlag.

[38] J. M. Pinilla, S. Finger, and F. B. Prinz. Shape feature description and recognition using an augmented topology graph grammar. In *NSF Engineering Design Research Conference*, pages 285–300. University of Massachusetts, Amherst MA, June 11-14 1989.

[39] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants.* Springer-Verlag, New York, 1990.

[40] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Developmental models of herbateous plants for computer imagery purposes. *Computer Graphics*, 22(4):141–150, 1988.

[41] A. A. G. Requicha. Mathematical models of rigid solids. Technical Report TM-28, Production Automation Project, University of Rochester, November 1977.

[42] A. A. G. Requicha. Mathematical foundations of constructive solid geometry: General topology of closed regular sets. Technical Report TM-27a, Production Automation Project, University of Rochester, 1978.

[43] A. A. G. Requicha. Representation of rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.

[44] A. A. G. Requicha and H. B. Voelcker. Solid modeling: Current status and research directions. *IEEE Computer Graphics and Applications*, 3(7):25–37, October 1983.

[45] A. A. G. Requicha and H. B. Voelcker. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 73(1):30–44, January 1985.

[46] D. J. Rosenkrantz. Programmed grammars: and new device for generating formal languages. In *8th IEEE Annual Symposium on Switching and Automata Theory*. IEEE, Austin, Texas, 1967.

[47] Daniel J. Rosenkrantz. Programmed grammars and classes of formal languages. *Journal of the Association for Computing Machinery*, 16(1):107–131, January 1969.

[48] George F. Simmons. *Introduction to Topology and Modern Analysis*. McGraw-Hill, New York, 1963.

[49] Robert F. Sproull. Parts of the frontier are hard to move. *Computer Graphics*, 24(2):9, March 1990.

[50] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.

[51] George Stiny. Ice-ray: a note on the generaation of Chinese lattice designs. *Environment and Planning B*, 4:89–98, 1977.

[52] George Stiny. Introduction to shape and shape grammars. *Environment and Planning B*, 7:343–351, 1980.

[53] George Stiny and William J. Mitchell. The Palladian grammar. *Environment and Planning B*, 5:5–18, 1978.

[54] George Stiny and William J. Mitchell. The grammar of paradise: on the generation of Mughul gardens. *Environment and Planning B*, 7:209–226, 1980.

[55] George Stiny and William J. Mitchell. A new line on drafting systems. *Design Computing*, 1:5–19, 1986.

[56] George Vanecek Jr. Protosolid: an inside look. Technical Report CSD-TR-921, Computer Science Department, Purdue University, November 1989.

[57] Kevin Weiler. *Topological Structures for Geometric Modeling*. PhD thesis, Rensselaer Polytechnic Institute, August 1986.